The MOSSIM Simulation Engine
Architecture and Design

William J. Dally

Computer Science Department
California Institute of Technology

Caltech-CS-TR-5123-84

The MOSSIM Simulation Engine

Architecture and Design

William J. Dally

Department of Computer Science
California Institute of Technology

5123:TR:84

April 1984

Abstract

As the complexity of VLSI circuits approaches 10^6 devices, the computational requirements of design verification are exceeding the capacity of general purpose computers. To provide the computing power required to verify these complex VLSI chips, special purpose hardware for performing simulation is required. Existing simulation engines which perform logic simulation are inadequate for MOS VLSI because they cannot accurately model MOS circuits. Switch-level simulation, on the other hand, models the affects of capacitance and transistor ratios at speeds comparable to logic simulation.

The MOSSIM Simulation Engine (MSE) is a special purpose processor for performing switch-level simulation of MOS VLSI circuits. A single processor MSE performs switch-level simulation 200 to 500 times faster than a VAX 11/780. Several MSE processors can be connected in parallel to achieve additional speedup. A virtual processor mechanism allows the MSE to simulate large circuits with the size of the circuit limited only by the amount of backing store available to hold the circuit description. Functional simulation is provided on the MSE to facilitate the efficient simulation of large circuits.

Contents

Plates

Chapter 1

Introduction

The complexity of VLSI circuits is increasing at an exponential rate [1] and will soon reach the level of 10^6 devices per chip. As the complexity increases, the computational requirements of computer-aided-design (CAD) tools used to design and test VLSI circuits will exceed the capacity of general purpose computers. Special purpose processors which can take advantage of the concurrency potential in CAD algorithms will be required to meet the computational requirements of designing VLSI circuits with 10^6 devices.

Design verification plays an essential role in the development of a VLSI chip. The complexity of the circuits, the inaccessibility of internal nodes, and the difficulty of repair make the probability of producing a working chip very low without extensive design verification. While techniques such as silicon compilation [2] offer the capability of verifying the correctness of a design by using a proven construction technique, conventional design techniques require extensive simulation to verify the correctness of a design. The burden of simulation is compounded by the fact that design is an iterative process. Each time a *bug* is discovered in the circuit and the design modified, all simulations must be repeated to verify that no additional *bugs* have been introduced by the modification. As circuits become more complex, special purpose simulation hardware is required to perform design verification in acceptable time.

A state of the art VLSI chip in 1982 contained $\approx 10^5$ devices and required about 2 weeks of CPU time on a VAX 11/780 to complete a single verification cycle. Since both the number of test vectors required to verify a chip and the amount of computation required to simulate one test vector scale at least linearly with the complexity of a chip, the amount of computation required to verify a chip scales at least quadratically with complexity. Thus, as shown in figure 1, a 1986 chip containing $\approx 10^6$ devices will require about 2 years of CPU time to completely verify on a conventional computer.

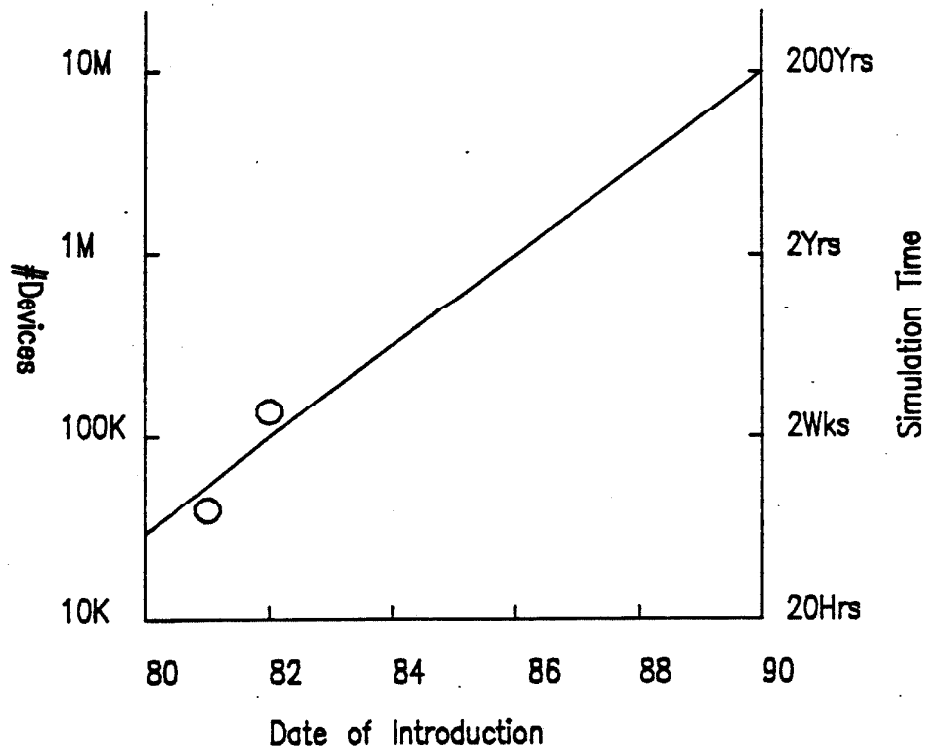


Figure 1. Scaling of Simulation Time

Conventional simulation engines such as the Yorktown Simulation Engine [3-5], ZyCAD Logic Evaluator [6], or TEGAS Accelerator [7], address this problem of long simulation times by performing logic simulation orders of magnitude faster than conventional computers. However, these logic simulation engines do not accurately model the behavior of MOS circuits. Simulation of charge sharing, precharged busses, sneak paths and other subtleties of MOS design are beyond the capabilities of these machines. Switch-level simulation is required to model these effects.

Bryant [8-11] has developed MOSSIM, a switch-level model and simulator which provides significantly more accurate simulation of MOS circuits than conventional methods of logic simulation [20]. By modeling MOS transistor ratios and node capacitances, and by considering a MOS transistor as a truly bidirectional device, MOSSIM provides simulation capabilities which previously required a circuit simulator [13,14] with simulation times comparable to logic simulation.

As simulation speeds increase, other steps in the design verification process become bottlenecks. The

reduction of hierarchical schematics databases to a format which can be loaded by the machine, and the transfer of input and output vectors to and from the machine must be accelerated to realize the benefits of hardware simulation.

This paper describes the preliminary architecture of a MOSSIM Simulation Engine (MSE) [15,16]. The goals of this architecture development were to design a machine which would:

1. Perform accurate simulation of MOS circuits.
2. Accelerate the speed of simulation by at least two orders of magnitude.
3. Scale well with increasing circuit complexity.
4. Support multi-level simulation allowing the simulation of leaf cells modeled at the MOS transistor level to be combined with the simulation of higher-level composition cells modeled at the register transfer or functional level.
5. Be extensible in terms of both features and network size.

The remainder of this paper describes the MSE in detail. Chapter 2 surveys previous work in the field of hardware simulation machines. Three machines are described and compared to the MSE. The MOSSIM algorithm [10] is presented in chapter 3. Extensions of the algorithm to support timing simulation and timing analysis are described. This chapter also examines the algorithms required to support circuit simulation and mixed mode simulation. Chapter 4 investigates the concurrency in these algorithms and develops the rationale for the MSE architecture. The architecture of the MSE at the register transfer level is presented in chapter 5. Chapter 6 discusses the implementation of this architecture in TTL technology. The results of simulations to determine the performance of the MSE are presented in chapter 7. Since the MSE performs the kernel of switch level simulation very rapidly, fast algorithms for compiling circuits from a schematics data base and feeding input/output vectors to these circuits are required, or these steps will become the bottleneck in the design verification process. Some algorithms for incremental network update and fast I/O are presented in chapter 8. Chapter 9 discusses the present status of the MSE project and the plans for future development. The MSE architecture is actually a general purpose network analyzer. This report concludes with a discussion of the application of this architecture to several problems.

Chapter 2

A Survey of Simulation Machines

A great deal of insight into the hardware simulation problem can be gained by examining the current state-of-the-art. The architecture of the MSE was to a great extent motivated by design ideas from existing simulation engines. Four simulation engines have been proposed to date and three of these engines have been constructed. In this chapter, the architectures of each of these machines is examined in terms of the goals outlined in chapter 1.

2.1 The Yorktown Simulation Engine (YSE) [3-5]

The Yorktown Simulation Engine is a multiprocessor composed of up to 256 *logic processors* each of which simulates a logic subnetwork containing up to 4K four-input gates. The logic processors are interconnected by a 256x256 crossbar switch which performs time-slot interchanging to line up matching logic signals. Each logic processor consists of an 8Kx128-bit instruction memory which controls the sequencing of a 4Kx2-bit data memory through an operation unit which simulates a general 4-input logic gate. The data memories are actually replicated five times to allow simultaneous access of the four inputs of the gate, and a value for the switch. A gate simulation consists of reading an instruction out of the instruction memory, fetching the four operands from the data memory, simulating the gate, storing the gate output back in memory, fetching the value to be exchanged with the switch, and sending that value to the switch. This sequence of operations takes 560ns to complete; however, a pipelining scheme is used which allows a new gate simulation to begin every 80ns. The YSE is closely related to the Logic Simulation Machine [17] built at IBM Los Gatos. Because of the similarity of the two machines only the YSE is described here.

The YSE performs either zero delay or unit delay logic simulation using an approach where every gate is simulated every cycle. This *compiled logic* [18] approach is inefficient as the activity in logic circuits

exhibits considerable locality and typically no more than 5%to 10%of the gates are active at any given time. The *compiled logic* simulation model used by the YSE also makes timing simulation very expensive to implement. A delay on the YSE is modeled by cascading several logic gates in series, each of which represents one unit delay. The advantage of this approach is speed. At 80ns per gate, a 256 processor YSE simulates circuits of up to 1M gates at a rate of 3.2 billion gate evaluations per second (GEPs), the fastest of any machine described in this chapter.

The YSE is also based on a unidirectional gate model which does not lend itself to MOS switch-level simulation. Although a switch level simulator has been *programmed* on the YSE, by devising logic circuits which model the relaxation equations of the switch-level model [10],[19]. This approach to switch level simulation, however, suffers from several limitations. First, since several logic gates are required to simulate a single transistor, the approach is inherently inefficient. Few values of strengths are available, making the modeling of phenomena such as charge sharing difficult. Also, the lack of an event-driven simulation kernel makes it impossible to determine when a relaxation has completed, forcing all relaxations to be run for a maximum time bound.

The software overhead to compile a circuit for simulation on the YSE is considerable. A circuit must first be partitioned into 4K blocks of gates so that the interconnections do not overload the capacity of the crossbar switch. Then the time slots for each logic gate are assigned and the logic interconnects are programmed into each logic processor's instruction memory. Initial values for the gate outputs are stored in the data memory. The switch must then be programmed to transfer the output of each logic processor with external connections to the input of the appropriate logic processor(s) with the appropriate time slot interchange. The elaborate compilation required by the YSE places a high cost on making changes in a circuit. Since the principal application of a logic simulation is to an iterative design process, a high cost on changes can be a serious detriment.

2.2 The ZyCAD Logic Evaluator [6]

The ZyCAD logic evaluator (ZLE) is the first commercially available simulation machine. While technical details of the machine have not been published, the available sales literature [6] indicates that it uses *event driven* [18] simulation. Only active logic gates are simulated during event driven simulation. As only 5%to 10%of the logic gates in a circuit are active at any one time, the ZLE is between 10 and 20 times more efficient than the YSE in this respect. The event driven simulation algorithm also supports timing simulation with little additional cost over unit-delay simulation. Like the YSE, the ZyCAD machine is limited to simulation of unidirectional logic gates making switch-level simulation difficult. A

three-valued strength system used by the ZLE provides some of the capabilities of a switch-level model but is extremely limited.

The ZLE is also a multiprocessor. Each processor simulates a subnetwork of up to 64K gates with a throughput of 2.5 million GEPs. Up to 15 processors may be connected on a bus for a total simulation capacity of 1M gates at a rate of 37.5 million GEPs. To compare this figure with the 3.2 billion GEPs of the YSE, however, we must take into account that the ZLE's *event driven* simulation is between 10 and 20 times more efficient than the YSE's *compiled logic* simulation. Correcting for this efficiency factor, the YSE would have a rating of 160 to 320 million event driven GEPs (EDGEPs), still almost an order of magnitude greater than the ZLE.

2.3 The TEGAS Accelerator [7]

The TEGAS Accelerator (TA) is planned as a speedup of the popular TEGAS simulator [20]. Like the ZLE, the TA is designed for *event driven* logic simulation. The architecture is designed to support simulation of up to 32K gates with throughput between 200K and 1M EDGEPs.

The TA exploits functional concurrency by dividing the simulation into evaluation and update phases. The operations taking place in each of these phases are distributed among eight function units, seven of which are active on each phase. However, only three of the function units are processors (the other five are memories), so at most two processors are operating concurrently.

The TA has the advantage of a large community of users who are already familiar with the TEGAS simulator; however, the architecture has some serious limitations. First, the upper limit of 32K gates is very restrictive. Perhaps a multiprocessor version is planned, but no indication of this is given in [7]. Also, the TA has no support at all for switch-level simulation. The TEGAS logic models are old and do not even have the *strength* capability supported by the ZLE. The TA does not make maximum use of the functional concurrency available in the event driven simulation algorithm. Only two processors in the TA are active at once. The Logic Simulation Machine described below keeps six processors active at once using a pipelined approach.

2.4 Abramovici Logic Simulation Machine [21-22]

Abramovici and co-workers at Bell Laboratories have described a Logic Simulation Machine (LSM) [21-22] which makes use of considerable functional concurrency. Like the ZLE, the LSM is designed for

event driven logic simulation. The architecture of the machine is a six stage pipeline where each stage represents one operation of the event driven simulation algorithm.

One unique feature of the LSM is that it supports functional simulation as well as logic simulation. Functional EValuators (FEVs) are used to simulate primitives considerably larger than a single logic gate such as a memory, counter, or ALU. Mixed mode simulation of this type provides significant speedup as the functional simulation requires much less time than simulation of the function unit at the gate level. This capability also matches the needs of a designer. Normally, the designer has a high degree of confidence in the operation of one subcircuit, A, while he is debugging another subcircuit, B. Mixed mode simulation offers him the capability of suppressing detail in A while B is debugged.

Unlike the YSE and ZLE, the LSM is not a multiprocessor. No partitioning by subnetwork is performed. Rather, speedup is achieved by microcoding the event driven simulation algorithm and pipelining its evaluation. The LSM has not been constructed; however, performance is estimated at a throughput of 500K EDGEPs.

2.5 Summary

The motivation in surveying the current state-of-the-art in simulation engines is to avoid repeating the work of others by copying the strengths in their designs while discarding the weaknesses. To analyze these strengths and weaknesses, each of the machines described above will be evaluated in terms of the five goals presented in chapter 1.

Accurate Simulation of MOS Circuits: As discussed above, none of the simulation machines is capable of accurate MOS switch-level simulation. While a switch level simulator has been implemented on the YSE [19], it has serious limitations. Adding strengths to logic values as is done in the ZLE is one approach to the problem; however, this approach is also limited in the situations it can model. To accurately simulate MOS circuits, the MSE must provide hardware support for switch-level MOS simulation. This capability will be achieved by implementing the MOSSIM algorithm [8-11] in hardware. The MOSSIM algorithm and related algorithms to be supported by the MSE are briefly described in chapter 3. The hardware to support the algorithm is described in chapters 5 and 6.

Simulation Speed: Five techniques for increasing simulation speed are used by the machines described above:

1. *Functional Concurrency* is achieved by performing several of the simulation functions in parallel. Examples of this technique in the machines described above are the pipelined gate evaluation

in the YSE and the pipelined event driven simulation algorithm in the LSM. In each of these cases, several functions (eg. fetch instruction, fetch data, simulate gate, store result,...) are performed simultaneously. The speedup gained by functional concurrency is equal to the number of concurrently executing units, n_f, multiplied by a utilization factor, u_f, which indicates the average percentage of units in simultaneous operation: $s_f = n_f u_f$. The utilization term includes factors such as pipeline loading and flushing overhead and idle time for units which are infrequently used. The application of functional concurrency to the MOSSIM algorithm is discussed in chapter 4.

2. *Subcircuit Concurrency* is achieved by partitioning the circuit into subcircuits and processing each subcircuit simultaneously. All of the machines described above make some use of subcircuit concurrency. As with functional concurrency, the speedup for subcircuit concurrency is the product of the number of units times the utilization: $s_s = n_s u_s$. However, in this case, the utilization term reflects the distribution of activity across the subcircuits, and possible contention in the interconnection mechanism. The low activity of logic circuits acts as a two-edged sword in this case. Since only 5%to 10%of the logic circuits are active at once, and since the active circuits tend to be clustered, only 5%to 10%of the processors may be active at once. This potential degradation of concurrency offsets many of the advantages of event-driven simulation. A virtual-processor mechanism which overcomes this problem is described in chapters 4 and 5.

3. *Specialization* involves dedicating special hardware to perform a frequently occurring operation. An example of specialization is the function unit in the YSE which simulates a general four input logic gate in one cycle. Without this special hardware, several cycles would be required to determine the gate output. Specialization can be thought of as a special case of functional concurrency, as what would normally be several operations take place at once. The distinction is that with specialization, the computation takes place in one unit rather than in several units operating concurrently. The expression for speedup due to specialization is again of the form: $s_p = n_p u_p$, where n_p is the number of cycles the operation would require without the unit divided by the number of cycles the operation requires with the unit, and u_p is the utilization factor of the unit. Special units for performing the relaxation operations of the MOSSIM algorithm are presented in chapters 4 and 5.

4. *Efficient Algorithms:* As illustrated by the comparison between *compiled logic* and *event driven* simulation, the use of an efficient algorithm offers a great potential for speedup. While the degree of speedup is dependent on the algorithms in question, the impact can in general be quite significant. The concurrency methods described above all improve throughput at best linearly (with the number of processors). A good algorithm on the other hand can improve throughput by a polynomial factor of the number of gates simulated. This makes efficient algorithms especially important for scaling

as discussed below. Algorithms for the MSE, and possible optimizations are discussed in chapter 3.

5. *Mixed Mode Simulation:* The capability to simulate *uninteresting* subcircuits at a high level gives a significant performance improvement. Much less computation is required to simulate one high-level unit (eg. a 64K RAM) than to simulate many smaller units and their interactions (eg. the 64K+ transistors in the RAM). So if a 64K RAM is simulated as a functional block, only 1.56×10^{-5} as many elements must be evaluated. Even if simulating the RAM requires 10 times the computation as simulating one transistor, the savings are still 1.56×10^{-4}. Event driven techniques may take some of the sting out of this number, simulating only one word-line and one-bit line of the RAM as well as associated decoders, sense-amplifiers and multiplexers would require about 10^3 transistors to be simulated, but the savings is still significant. Mixed mode simulation also greatly increases the capacity of the machine. Simulating functional blocks at a high level leaves more of the gate capacity of the machine available for the rest of the circuit.

   Both the YSE and the LSM provide some level of mixed mode simulation. The YSE's *array processors* simulate memories at a high level. A more complete functional simulation capability is provided by the LSM's *functional evaluators (FEVs)*. Mixed mode simulation on the MSE is discussed in chapters 3, 4 and 5.

Scaling: A major motivation for simulation machines is to relieve general purpose computers of the computational burden of simulation as circuits approach the complexity of 10^6 devices. To achieve this goal, a simulation machine must scale well both in terms of the upper limit on the number of gates/transistors which can be simulated and in terms of the performance degradation with increasing complexity. All of the simulation machines described above have an upper limit on the number of gates which can be simulated. For instance, both the YSE and the ZLE can simulate 1M gates. While these limits may seem reasonable today, tomorrow's VLSI circuits may not fit on these machines. The virtual processor concept, and the hierarchical addressing mechanism of the MSE, described in chapters 4 and 5 are designed to overcome an upper limit on simulation capacity.

Mixed mode simulation as described above offers another means of increasing the capacity of a machine. By making use of hierarchical models and simulating only a small portion of a circuit at the lowest level, very large systems may be simulated with a small gate/transistor capacity. A second advantage of this approach is that there is very little degradation in performance with increasing complexity. The mixed mode approach to increasing capacity, while it is a useful capability, lacks generality. There are some cases when the simulation of an entire circuit at the transistor level is required.

When a *flat* simulation of an entire (large) circuit at the transistor level is required, the only means

of preventing performance degradation is subcircuit concurrency. As more processors are added to a simulation machine, extreme demands are placed on the interconnection network. The scaling issue as it applies to interconnection networks is discussed in chapter 4.

Extensibility: All of the machines discussed in this chapter are strictly logic simulation machines. Also, the set of features which these machines support in their simulations is fixed by the hardware architecture. While it is possible to *program* these machines in an awkward manner such as in [19], or to reprogram a microcoded machine such as the LSM, programming new applications on existing hardware is awkward at best.

An examination of the MOSSIM simulation algorithm indicates that it consists mainly of performing simple manipulations on a network data base. It would appear that several other applications could be made to operate on the same data base such as timing simulation, timing verification, circuit simulation, functional simulation and network partitioning. Algorithms for each of these applications are discussed in the next chapter.

Chapter 3

Algorithms

The MSE architecture is designed to efficiently implement the MOSSIM algorithms while at the same time providing a framework which can be extended to include timing simulation, circuit simulation, and functional block simulation. These algorithms form the starting point for the development of the architecture by motivating the data structures and operations which must be implemented in hardware to improve performance. At the same time, the constraints of hardware implementation suggest modifications to the algorithms which greatly simplify the amount of hardware required.

In this chapter the simulation algorithms are discussed. We begin by describing the MOSSIM algorithm. This is the only algorithm implemented on the prototype MSE; however, architectural hooks are included in the architecture to allow the implementation of other algorithms at a later date. Bryant's switch-level model [10] is the basis of switch level simulation. In section 3.1 we describe the model and how it is mapped into a data structure which is efficiently implemented and manipulated in hardware. This data structure differs significantly from the data structure used in a software implementation of MOSSIM. Next, in section 3.2, the MOSSIM algorithm is presented. This algorithm is then manipulated so that each part of the algorithm fits into an *algorithm template* allowing them to share the same hardware. In section 3.3, an extension of the MOSSIM algorithm to timing simulation based on a timing model developed by Lin and Mead [23] is discussed. The use of the MSE architecture for circuit simulation is discussed in section 3.4. A goal of the MSE is to perform mixed-mode simulation. This mixed mode capability is based on a hardware implementation of the DSIM functional simulator [24]. The algorithms used by this simulator are discussed in section 3.5. It is also shown that functional simulation can be used to relieve the I/O vector bottleneck associated with hardware simulators.

3.1 Switch Level Model

The switch level network model consists of a set of *nodes* $N = \{n_1, \ldots, n_n\}$ connected by a set of *transistors* $T = \{t_1, \ldots, t_m\}$. The capacitance of nodes is modeled by assigning each node a *size* , κ, from the set $K = \{\kappa_1, \ldots, \kappa_{max}\}$ according to the relative capacitance of the node compared to other nodes in the network. Node sizes are ordered, $\kappa_1 < \cdots < \kappa_{max}$, but have no other properties. Input nodes are assigned size $\omega > \kappa_{max}$. The abstraction of node *size* accurately models the behavior of charge sharing in MOS circuits while avoiding the complexity of parametric capacitances.

Transistor *strength* is an abstraction which models the effect of transistor ratios in the circuit. Each transistor is assigned a strength, γ, from the set $\Gamma = \{\gamma_1, \ldots, \gamma_{max}\}$ where the only property of transistor strengths is their ordering, $\gamma_1 < \cdots < \gamma_{max}$. The strength of a transistor is loosely related to its on-conductance (proportional to ratio) in the same manner that node size is loosely related to capacitance.

Each node, n_i, has a state, y_i, and each transistor, t_j, has a state, z_j, from the set $\{0, 1, X\}$. Node state, y_i, represents the node voltage with 0 and 1 corresponding to low and high voltages, and X representing either a voltage between low and high, or an unknown voltage. The state of a transistor, z_j, indicates whether the transistor is on or off (0-off, 1-on, X-unknown).

Each terminal of a transistor (source, gate and drain) is connected to a node. When a transistor is on, the source and drain nodes are connected by a resistor with conductance approximanted by the transistor's strength. The state of the transistor is controlled by the node connected to its gate. The following table gives the relation between gate node state and transistor state for each type of transistor modeled.

GATE STATE	0	1	X
N-TYPE	0	1	X
P-TYPE	1	0	X
D-TYPE	1	1	1
O-TYPE	0	0	0

Table 1. Transistor State vs. Gate State

Signal strength is a measure of how much current the signal present on a node can deliver. It can be thought of as the driving capability of a signal. Signal strengths are elements of the set $\{\lambda\} \bigcup K \bigcup \Gamma$. If a signal has no path to a source of current or charge, it is of strength λ. A signal with strength κ_i is totally isolated from current sources and can supply current only from the charge stored on a node. Signals with strength ω are directly connected to inputs and are capable of supplying a very large

amount of current. Signals which are driven by an input through a chain of *on* transistors have strength equal to the maximum over all paths to the input of the minimum transistor strength on a path, γ_j. The ordering of signal strengths is $\lambda < \kappa_1 < \cdots < \kappa_{max} < \gamma_1 < \cdots < \gamma_{max} < \omega$. This ordering reflects the fact that *input* nodes overdrive any node which is *driven* through a transistor, and nodes which are driven over-drive any node which is *charged*.

In finding the new state of a network, the MOSSIM algorithm solves a set of recurrence equations for three strengths associated with each node. An algebraic formulation leading to the equations and a description of each of the strengths is given in [10]. In this paper we will refer to these strengths as blocking-strength, r_i, the up-strength, u_i, and the down strength, d_i. The blocking strength of a node is the maximum signal strength which can reach a node through transistors which are definitely on (state = 1). (Note that when a signal propagates through a transistor, the strength on the destination side of the transistor is the minimum of the signal strength on the source side of the transistor and the transistor strength.) The up (down) strength of a node is the maximum logic 1 (0) signal strength which can reach the node through any transistor which can possibly be on (state = 1 or X).

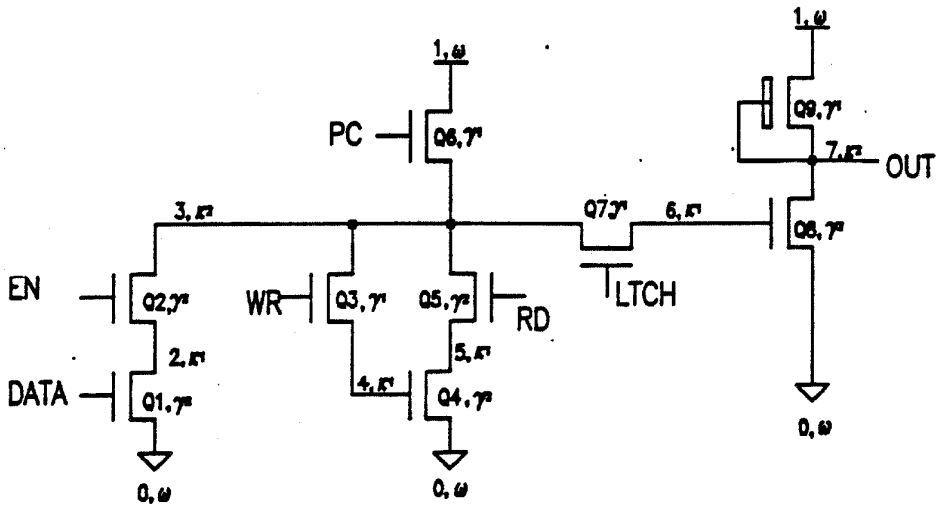


Figure 2. Example Circuit

As an example of the switch level model, consider the example circuit shown in figure 2 above. The circuit consists of a three-transistor memory cell on a precharged bus. The bus can be pulled low from an input or latched onto the input of an inverter. Each transistor is modeled with a strength; transistors $Q_{1,2,4,5,8}$ have strength γ_2 while transistors $Q_{3,6,7,9}$ have strength γ_1. Transistors Q_{1-8} are N-type while Q_9 is P-type. Each circuit node is numbered and assigned a size. Nodes 0 and 1 are V_{dd} and GND with size ω, nodes 2,4,5 and 6 have size $\kappa 1$, and nodes 3 and 7 have size $\kappa 2$. Note that since the input of the inverter, node 6, has a smaller size than the bit line, node 3, there is no chance of charge sharing between these nodes. This circuit will be used in the next sections to show an example data structure and to show the algorithm in operation.

Data Structure The MOSSIM algorithm operates on a data structure which describes the network model. The four major data structures used by the MSE algorithm are the *node list*, the *link list*, the *gate list*, and the *event stacks*.

The *node list* contains a record for each node in the network. Each record contains the following fields:

R	U	L	P	NSTR	RSTR	USTR	DSTR	LS	NLS	LP	GP

1. **R, U, L and P Activity Flags:** These flags are set to indicate when the corresponding operation is scheduled for the node. The correspondence between flag and operation is described in the following table:

FLAG	OPERATION
R	BLOCKING STRENGTH RELAXATION
U	UP/DOWN STRENGTH RELAXATION
L	LOGIC UPDATE
P	PERTURBATION

Table 2. Activity Flags

2. **Node Size and Signal Strengths:** The next four fields of the node record contain the size of the node **NSTR**, and the current value of the three relaxation strengths, **RSTR, USTR, DSTR**. The size and strength fields each contain four bits which are interpreted as follows:

CODE	STRENGTH
0000	λ
0001	κ_1
⋮	⋮
x	κ_{max}
$x+1$	γ_1
⋮	⋮
1110	γ_{max}
1111	ω

Table 3. Strength Codes

The strengths are encoded in such a manner that the total number of node sizes (excluding inputs), n_κ, plus the total number of transistor strengths, n_γ, must equal 14. The x in the table corresponds to $n_\kappa + 1$.

3. **Logic States:** Two fields, **LS** and **NLS**, are used to hold the old and new logic states respectively. The logic states are encoded as follows:

CODE	STATE
00	LO
01	HI
10	X

Table 4. Logic State Codes

4. **Link Pointer:** The link pointer, **LP**, points to the first entry of the link list for this node.

5. **Gate Pointer:** The gate pointer, **GP**, points to the first entry of the gate list for this node.

The *link list* contains a record for each *link* in the network. A *link* indicates the connection of two nodes by the channel of an MOS transistor. Each transistor is modeled by two links: one link from the source to the drain, and one link from the drain to the source. The link list is actually made up of a number of sublists each of which contains the links corresponding to one node. The **LP** field of the node record points to the first link record of the sub-list corresponding to that node. Each sublist is terminated by a link of type Null or Last. In dealing with links, the node pointing to the link list will be referred to as the *source node* and the link pointed at by a link's **NP** field will be considered the *destination node.*

A link record contains the following fields:

TS	TSTR	LTYP	TTYP	NP

1. **Transistor State:** The **TS** field represents the current transistor state according to the following table:

CODE	STATE
00	OPEN
01	CLOSED
10	X

Table 5. Transistor State Codes

2. **Transistor Strength:** The transistor strength, coded identically to the node strengths as shown in table 3, is contained in the **TSTR** field.

3. **Link Type:** The link type **LTYP** field determines how the link record is interpreted. The four possibilities are illustrated in the following table:

CODE	LINK TYPE
00	NULL LINK
01	INTERNAL LINK
10	LAST LINK
11	EXTERNAL LINK

Table 6. Link Type Codes

4. **Transistor Type:** The **TTYP** field indicates the type of transistor according to the following table. The transistor types are defined in table 1.

CODE	TRANSISTOR TYPE
00	N-TYPE
01	P-TYPE
10	D-TYPE
11	O-TYPE

Table 7. Transistor Type Codes

5. **Node Pointer:** The **NP** field contains a pointer to the destination node of this link.

The *gate list* represents the gate connections of transistors. Each record in the gate list connects a gate node with a link it controls. A gate record has only two fields.

GTYP	LP

1. **Gate Type:** The **GTYP** field of the gate record instructs the MSE processor how to interpret the record. This field is coded identically to the link type field shown in table 6.

2. **Link Pointer:** The **LP** field points to the link controlled by this gate record.

The *event stacks* are used for scheduling the simulation operations on nodes. There are four sets of stacks which correspond exactly to the activity flags described in table 2. The blocking strength relaxation stack set **RSTK** and the up/down strength relaxation stack set **USTK** each consist of sixteen separate stacks. One stack is used for each value of strength. The rationale for separate stacks is to *relax* the nodes in decreasing order of strength. As shown in [10], this ordering of relaxation significantly reduces the number of relaxations which must be performed. The logic update stack set **LSTK** and the perturbation stack set **PSTK** each contain only a single stack as the order of update does not influence the number of operations to be performed.

Example: Consider the example circuit of figure 2 above. The node list, link list and gate list for this

circuit and their possible initial states are shown in tables 8,9 and 10 below. Note that the RSTR, USTR, DSTR, LS and NLS fields of the node records and the TS field of the link record represent the state of the network while all other fields represent the network itself and do not change during simulation. The activity flags have been omitted. Also, the input nodes controlling the gates of $Q_{1,2,3,4,6,7}$ are not listed in the node list. Input nodes which connect only to gates of transistors need not be entered into the circuit as nodes. A changing input of this type directly causes the link records corresponding to its gate list to be updated.

NODE	NSTR	RSTR	USTR	DSTR	LS	NLS	LP	GP
0	ω	ω	λ	ω	0	0	1	0
1	ω	ω	ω	λ	1	1	4	0
2	κ_1	λ	λ	λ	X	X	6	0
3	κ_2	λ	λ	λ	X	X	8	0
4	κ_1	λ	λ	λ	X	X	13	1
5	κ_1	λ	λ	λ	X	X	14	0
6	κ_1	λ	λ	λ	X	X	16	0
7	κ_2	λ	λ	λ	X	X	17	3

Table 8. Example Node List

LINK	Q	TS	TSTR	LTYP	TTYP	NP
1	1 ↑	X	γ_2	INTERNAL	N-TYPE	2
2	5 ↑	X	γ_2	INTERNAL	N-TYPE	5
3	8 ↑	X	γ_2	LAST	N-TYPE	7
4	6 ↓	X	γ_1	INTERNAL	N-TYPE	3
5	9 ↓	1	γ_1	LAST	D-TYPE	7
6	1 ↓	X	γ_2	INTERNAL	N-TYPE	0
7	2 ↑	X	γ_2	LAST	N-TYPE	3
8	2 ↓	X	γ_2	INTERNAL	N-TYPE	2
9	3 ↓	X	γ_1	INTERNAL	N-TYPE	4
10	4 ↓	X	γ_2	INTERNAL	N-TYPE	5
11	6 ↑	X	γ_1	INTERNAL	N-TYPE	1
12	7 →	X	γ_1	LAST	N-TYPE	6
13	3 ↑	X	γ_1	LAST	N-TYPE	3
14	5 ↓	X	γ_2	INTERNAL	N-TYPE	0
15	4 ↑	X	γ_2	LAST	N-TYPE	3
16	7 ←	X	γ_1	LAST	N-TYPE	3
17	8 ↓	X	γ_2	INTERNAL	N-TYPE	0
18	9 ↑	1	γ_1	LAST	N-TYPE	1

Table 9. Example Link List

GATE	Q	GTYP	LP
1	5 ↑	INTERNAL	2
2	5 ↓	LAST	14
3	9 ↓	INTERNAL	5
4	9 ↑	LAST	18
5	8 ↑	INTERNAL	3
6	8 ↓	LAST	17
$\overline{DATA}$	1 ↑	INTERNAL	1
$\overline{DATA}$	1 ↓	LAST	6
EN	2 ↑	INTERNAL	7
EN	2 ↓	LAST	8
WR	3 ↓	INTERNAL	9
WR	3 ↑	LAST	13
RD	4 ↓	INTERNAL	10
RD	4 ↑	LAST	15
PC	6 ↓	INTERNAL	4
$\overline{PC}$	6 ↑	LAST	11
$LTCH$	7 →	INTERNAL	12
$LTCH$	7 ←	LAST	16

Table 10. Example Gate List

3.2 The MOSSIM Algorithm

The primary function of the MOSSIM Simulation Engine (MSE) is to improve the performance of MOS Switch-Level Simulation using the MOSSIM algorithm developed by Randy Bryant [8-11]. This algorithm, described in detail in [10], is based on a formal switch-level model of MOS transistor networks. By modeling MOS transistor ratios and node capacitances and by considering an MOS transistor as a truly bidirectional device, MOSSIM provides considerably more accurate simulation of MOS LSI than conventional logic simulators [20]. MOSSIM achieves performance comparable with logic gate simulators by using an incremental algorithm which exploits the sparseness and locality of events in a circuit.

Now that we have defined a network model and encapsulated it in a data structure, we are prepared to discuss the MOSSIM algorithm. The MOSSIM algorithm as adapted for the MSE hardware implementation performs repeated solutions of the steady state excitation of the network until a stable state is reached. Each steady state excitation is arrived at by finding the strongest unblocked path pulling up and pulling down to each node. The node's state is set by comparing the strengths of these paths [10].

```
Find initial state of network
For each input vector
    Update input nodes
        Incrementally find new state of network
            1. Logic Update
            2. Perturbation
            3. Blocking Strength Relaxation
            4. Up/Down Strength Relaxation
```

MOSSIM ALGORITHM - MAJOR LOOP

The incremental network solution proceeds iteratively in four steps (1-4) until the network reaches a stable state or until oscillation is detected:

1. *logic update:* Update the state of each link whose gate node is on the **LSTK**. Schedule the *destination node* of each link whose state changes on the **PSTK**.

2. *perturbation:* Schedule the recalculation of strengths, r_i, u_i, d_i, for nodes which are in the *vicinity* of each node on the **PSTK**. Nodes are rescheduled on both the **RSTK** and the **USTK**.

3. *blocking strength relaxation:* Find the blocking signal strength, r_i, for each node on the **RSTK**. The blocking strength is the strength of the strongest path which definitely reaches a node.

4. *up/down strength relaxation:* Find the up and down signal strengths, u_i, d_i, and the logic state, $y_i = f(r_i, u_i, d_i)$, for each node on the **USTK**. Schedule any node whose logic state changes on the **LSTK**.

Since the process is incremental, step 1 updates only those nodes which have been scheduled in step 4 of a previous cycle. In fact, each step is event driven. Step 4 schedules the logic update events (transistor gate changes) to be processed by step 1. Step 1 schedules the perturbation events for step 2. Step 2 schedules the relaxation events for both step 3, and step 3 schedules the relaxation events for step 4. The initial scheduling is performed by finding those nodes which are in the vicinity of input nodes which change state, or in the vicinity of the source or drain nodes of transistors whose gate nodes are inputs which change state. The steady state is found when no further relaxations are scheduled by step 4.

Although the monotonic nature of the relaxation processes guarantees convergence [10], it is possible that there is no steady logic state for the network. (For example, consider a ring oscillator.) To prevent

the endless iteration of the four steps, some type of oscillation detection algorithm such as described in [18] is required to halt the process if it does not converge.

Logic Update: During the Logic Update (LU) step, the state of all transistors whose gate node changed state in the last iteration are updated. The list of links to be updated is sequenced by a node's gate list. For each transistor gate controlled by a node, the `l.TS := f(n.NLS, l.TTYP)` operation sets the transistor state as a function of the node state and transistor type according to table 1. If the link, `l = link[G.lp]` has changed state as a result of this operation, the node, l.NP is scheduled for the perturbation step.

```
For each node, n, on the LSTK
    For each gate record, g, in the gate list of n
        l := link[g.LP]
            l.TS := f(n.NLS,l.TTYP)
            if l.TS has changed, push l.NP on PSTK
```

LOGIC UPDATE ALGORITHM

Perturbation: The nodes which may be affected by each transistor changed in the logic update step are found during the Perturbation step (P). This step resets the signal strengths of these nodes and schedules them for relaxation. Each node scheduled on the perturbation stack has its strengths set as if the node were charged, and is scheduled for blocking strength relaxation. Since node N changing can affect the state of adjacent nodes, all nodes attached to N by active transistors, `l.NP`, are then scheduled for perturbation. This search for adjacent nodes finds the *vicinity* of the perturbed node and schedules it for relaxation.

```
For each node, n, on the PSTK
    n.RSTR := min(n.NSTR,n.RSTR)
    n.USTR := min(n.NSTR,n.USTR)
    n.DSTR := min(n.NSTR,n.DSTR)
    schedule n on the RSTK and USTK
    For each link, l, in n.LP, with l.TS = 1 or l.TS = 2
        m := node[l.NP]
        if m is not on the PSTK or the RSTK
```

```
schedule m on the PSTK
```

PERTURBATION ALGORITHM

Since a link connected to a perturbed node has changed state, the node's strengths are no longer valid and must be recomputed. To recompute the strength values, a perturbed node's strengths are reset to their *charged* levels and the node is rescheduled for relaxation. The new values of the strengths are then computed during the following relaxation steps.

If the state of the link causing the perturbation is examined, two optimizations can be made for closed or unknown links. First, if the link state is closed, the blocking strength **RSTR** need not be reset, as it can only be increased by a link closing. Also, only one of a complementary pair of links need perturb its destination node, as the other node will be perturbed through the link.

Further optimizations are possible if ternary simulation [10] is used. Ternary simulations involve two phases. During phase 1, all changing inputs are set to X and the circuit is simulated until steady state is reached. Then, during phase 2, the changing inputs which were set to X in phase 1 are set to their new state and the circuit is again simulated until steady state is reached. Since these input changes are monotonic in nature, the changes to node strengths are also monotonic. During phase 1, **USTR** and **DSTR** can only increase, so they need not be reset during perturbation. Blocking strength, however, must be reset during phase 1. The situation is reversed during phase 2. Since transistor states are becoming more definite, blocking strength can only increase and need not be reset during perturbation. **USTR** and **DSTR** however, may decrease and must be reset. These optimizations can reduce the amount of time required for perturbation and the subsequent relaxations by as much as 50%. Neither of these optimizations is currently performed in the MSE.

Blocking Strength Relaxation: During the Blocking Strength Relaxation (BSR) step, the strength of the strongest definite path to a node is found using a modified path tracing algorithm [25]. Each node, n, scheduled for relaxation is removed from the stack in order of strength, the strongest nodes first. This node is scheduled for up/down relaxation. Node n is then *relaxed* against all nodes, m, connected by definitely on transistors. The blocking strength of m is set to be: $max(min(str(l), Bstr(n)), Bstr(m))$. If the blocking strength of m changes, m is scheduled for BSR.

```
For each node, n, scheduled on RSTK
    schedule n on USTK
```

```
For each link, l, in n.LP, with l.TS = 1
    m := node[l.NP]
    m.RSTR := max(m.RSTR,min(n.RSTR,l.TSTR))
    If m.RSTR changed
        schedule m on RSTK[m.RSTR]
```

BLOCKING STRENGTH RELAXATION ALGORITHM

This algorithm sets the blocking strength of each scheduled node by *relaxing* each node against all those nodes connected to it by a closed transistor. The blocking strength **RSTR** of the destination node is updated to reflect the maximum strength which will definitely reach that node. The nodes are relaxed in order of decreasing strength by removing nodes from the higher numbered **RSTK**s first. As a node is removed from the stack, its **R** activity flag is reset to indicate that it is no longer scheduled for a blocking strength relaxation. Any node whose strength changes as a result of a relaxation operation is scheduled for relaxation. Note that a node can be scheduled on a given stack set more than once, since it is scheduled each time its strength changes. However, it will only be relaxed once since if it's R flag is reset when it is popped off the stack relaxation is not performed.

Up/Down Relaxation:

```
For each node, n, scheduled on USTK
    For each link, l, in n.LP, with l.TS = 1 or l.TS = 2
        m := node[l.NP]
        If min(n.USTR,l.TSTR) > m.RSTR
            m.USTR := max(m.USTR,min(n.USTR,l.TSTR))
        If min(n.DSTR,l.TSTR) > m.RSTR
            m.DSTR := max(m.DSTR,min(n.DSTR,l.TSTR))
        If m.USTR or m.DSTR changed
            schedule m on USTK[max(m.USTR,m.DSTR)]
            If m.USTR >= m.RSTR > m.DSTR
                m.NLS = hi
            Else If m.DSTR >= m.RSTR > m.USTR
                m.NLS = lo
            Else
                m.NLS = X
```

```
If n.NLS <> N.LS
    schedule n on LSTK
```

UP/DOWN STRENGTH RELAXATION ALGORITHM

The Up/Down strength relaxation algorithm is similar in form to the blocking strength relaxation algorithm. Both algorithms sequence nodes for relaxation in the same manner and differ only in the relaxation operation performed and in the scheduling for logic update. For each node scheduled, the destination node of every link which is in the closed or unknown state is updated. The destination node has its up strength **USTR** and down strength **DSTR** fields updated to reflect the largest up/down strength to reach that node without being *blocked* by a greater value of blocking strength **RSTR**. The new logic state of the node **NLS** is set according to the values of the three strengths. If exactly one of the up/down strengths is greater than or equal to the blocking strength, the logic state is set to the corresponding logic value (HI for up, LO for down); otherwise, the new logic state is set to X. Any node which has a strength change is scheduled for relaxation on the **USTK** set, and any node which has a logic state change is scheduled for logic update on the **LSTK**.

To develop an adaptation of the MOSSIM algorithm for the MSE, each of the four steps was manipulated to fit into the same algorithm template.

```
For each node, N, scheduled
  Operate on N
  if update1 schedule N for next step
  For each active link, L, (gate, G,)
  attached to N
    Operate on dest[L] (dest[G])
    if update2 schedule dest[L] for this step
    if update3 schedule dest[L] for next step
```

MSE - ALGORITHM TEMPLATE

Each step is identical except for 1) the operation performed 2) the definition of an active link, 3) the update criteria for scheduling nodes, and 4) as the parenthesis indicate, for the LU step, the operation is performed on a transistor rather than a node. The following table gives the operations and update

conditions for each of the four steps. This factoring of code across the simulation steps allows the same hardware to be used for each simulation step.

Op	Logic Update	Perturbation	Blocking Relaxation	Up/Down Relaxation
Operate on N	LS ← NLS	Reset STRs	Reset R bit	Reset U bit
Update 1	FALSE	TRUE	TRUE	FALSE
Operate on Dest	None	l.TS ← f()	RRELAX	U/D RELAX
Update 2	FALSE	TRUE	change(RSTR)	change(USTR,DSTR)
Update 3	change(l.TS)	FALSE	FALSE	LS ≠ NLS

Table 11. Algorithm Template Functions

Example: A simple example illustrates some of the subtle details of the MOSSIM Algorithm. Consider the circuit of figure 2 above. The following test vectors will be applied to the circuit.

EN	$\overline{DATA}$	PC	WR	RD	LTCH	Comment
0	0	0	0	0	0	Initialize
1	1	1	0	0	1	Discharge bus, illustrate blocking
1	1	0	0	0	1	Intermediate step
0	0	0	1	0	1	Write a zero, illustrate charge sharing

Table 12. Example Test Vectors

Initially the nodes are in the unknown state except node 7 which is a 1. At the beginning of a simulation all nodes are scheduled on the logic update stack so the entire network gets simulated. The first test vector turns off all externally controlled transistors leaving the connecting nodes in their initial states. Transistors Q_8 and Q_5 are in an unknown state making nodes 7 and 5 driven unknown.

NODE	RSTR	USTR	DSTR	LOGIC STATE
2	κ_1	κ_1	κ_1	X
3	κ_2	κ_2	κ_2	X
4	κ_1	κ_1	κ_1	X
5	κ_1	κ_1	γ_2	X
6	κ_1	κ_1	κ_1	X
7	γ_1	γ_1	γ_2	X

Table 13. Node States after Vector 1

In the logic update step following the second test vector, the gates of transistors $Q_{1,2,6,7}$ go to the high state and these transistors are turned on. Nodes 2,3 and 6 are pushed on the perturbation stack since they are connected to the changing transistors. Note that input nodes are not pushed on the perturbation stack.

In the perturbation step, these nodes are set to the charged to X state, USTR = DSTR = RSTR = κ.

During blocking strength relaxation, nodes 2 and 3 get a blocking strength of γ_2 while node 6 gets a blocking strength of γ_1. During up/down relaxation, the γ_2 blocking strength on node 3 *blocks* the path from V_{dd} through the precharge transistor to node 6. Thus, node 3 and its connecting nodes, 2 and 6, get an up strength of λ. This blocking is one of the more subtle aspects of MOSSIM. Even though both V_{dd} and GND reach node 6 through a series of transistors with minimum strength γ_1, the path from V_{dd} is blocked preventing node 6 from being set to an unknown state.

Since node 6 changes state from X to 0, it is pushed on the logic update stack and a second iteration through the four simulation steps is required. In the second logic update step, transistor Q_8 changes state from X to 0, and node 7 gets pushed on the perturbation stack. Node 7 gets reset to RSTR = USTR = DSTR = κ_2, and then during the relaxation steps RSTR and USTR get set to γ_1 because Q_9 is conducting.

It would appear that since node 7 changed state another iteration would be required to update the state of transistor Q_9, however, since D-type transistors are always on, no logic update is required.

The node states after the second test vector are shown below.

NODE	RSTR	USTR	DSTR	LOGIC STATE
2	γ_2	λ	γ_2	0
3	γ_2	λ	γ_2	0
4	κ_1	κ_1	κ_1	X
5	κ_1	κ_1	γ_2	X
6	γ_1	λ	γ_1	0
7	γ_1	γ_1	λ	1

Table 14. Node States after Vector 2

In the third test vector, the precharge transistor gets turned off. This has no effect on the strengths since all paths from Q_6 were blocked by the γ_2 DSTR on node 3.

The fourth test vector turns off Q_1 and Q_2 removing all drive from node 3, and turns on Q_6 writing a zero onto node 4. Examining this step in more detail, the test vector pushes nodes ($\overline{DATA}$,EN,WR) onto the logic stack, and the simulator then processes these nodes as follows:

1. *Logic Update:* Stack = ($\overline{DATA}$,EN,WR).

   a. $\overline{DATA}$ is popped and set to its new state, 0. The gate list associated with this node is then

scanned setting the state of links 1 and 6 to 0 and scheduling node 2 for perturbation.

b. *EN* is popped and set to state 0. Controlled links 7 and 8 are set to state 0 and node 3 is scheduled for perturbation. (Node 2 is already scheduled so it is not pushed.)

c. *WR* is popped and set to state 1. Controlled links 9 and 13 are set to state 1, and node 4, pointed to by link 9, is pushed.

2. *Perturbation:* Stack = (4,3,2).

a. Node 4 is popped and set to *charged to X*, RSTR = USTR = DSTR = κ_1. Node 3 is connected by an active link to node 4, but since it is already on the stack it is not rescheduled.

b. Node 3 is popped and set to *charged to 0*, RSTR = DSTR = κ_2, USTR = λ. Nodes 4 and 6 are connected by active links to node 3. Since node 6 is not yet on the stack it is pushed.

c. Node 2 is popped and set to *charged to 0*, RSTR = DSTR = κ_1, USTR = λ.

3. *Blocking Relaxation:* Since perturbation pushes every node, Stack = $[\kappa_2{:}(3),\kappa_1{:}(4,2,6)]$.

a. Node 3 is popped. Its link list beginning at link 8 is scanned for active links. Links 9 and 12 pointing to nodes 4 and 6 are active, so node 3 is relaxed against node 4 through link 9 and against node 6 through link 12. The blocking strength of nodes 4 and 6 are set to κ_2 from these relaxations and since they changed state both are pushed on the κ_2 RSTK. Node 3 is pushed on the USTK with strength κ_2.

b. Node 4 is popped and relaxed through link 13 against node 3. There is no change. Node 4 is pushed on the USTK with strength κ_2.

c. Node 6 is popped and relaxed through link 16 against node 3. There is no change. Node 6 is pushed on the USTK with strength κ_2.

d. Node 4 is popped and relaxed against adjacent node 3. There is no change as node 3 has RSTR = κ_2 and node 4 has RSTR = κ_1. Node 4 is pushed on the USTK with strength κ_1.

e. The κ_2 RSTK is empty, so node 4 is popped off the κ_1 stack. Since it has already been relaxed, it is ignored.

f. Node 6 is popped off the κ_1 stack and ignored.

g. Node 2 is popped. Both links in its link list 6 and 7 are off so no relaxations are performed.

Node 2 is pushed on the USTK with strength $kappa_1$.

4. Up/Down Relaxation: Stack = [κ_2:(6,4,3),κ_1:(2)].

   a. Node 6 is popped and relaxed through link 16 against node 3. Node 6 up and down strengths κ_1 are blocked by κ_2 RSTR at node 3.

   b. Node 4 is popped and relaxed through link 13 against node 3. All paths are blocked.

   c. Node 3 is relaxed through links 9 and 12 against nodes 4 and 6 changing both nodes to DSTR = κ_2 and Logic state = 0. Since node 4 changed state, it is scheduled on the LSTK.

5. *Second Iteration:* Omitting the details, during the second iteration transistor Q_5 changes state, node 5 is perturbed, but since it has no active links the change does not propagate.

The final state of the nodes is shown below. Note that the MOSSIM algorithm correctly decided the charge sharing conflict between nodes 3 and 4 on the basis of node size. This example also illustrates how MOSSIM accurately models stored charge on nodes, something which cannot be done with most logic simulators. Also note that by maintaining stacks ordered by strength, a node was never relaxed more than once.

NODE	RSTR	USTR	DSTR	LOGIC STATE
2	κ_1	λ	κ_1	0
3	κ_2	λ	κ_2	0
4	κ_2	λ	κ_2	0
5	κ_1	κ_1	κ_1	X
6	κ_2	λ	κ_2	0
7	γ_1	γ_1	λ	1

Table 15. Node States after Vector 4

3.3 Timing Simulation and Timing Verification

The proper operation of a digital system requires not only that the system is *logically* correct, but also that the system timing is correct. While the MOSSIM algorithm using ternary simulation can detect such timing errors as races and hazards, it has no means of checking that path delays are not excessive. Lin and Mead [23] have developed a delay model suitable for application to switch-level simulation.

This section gives a brief description of the delay model and discusses its applications to timing simulation

and timing verification on the MSE. A switch-level timing simulator assigns specific times to each logic value transition. The simulation results are used to verify that the circuit operates properly with *approximate* signal delays. One weakness of this approach to timing verification is that it is very difficult to come up with a set of test vectors which will exercise every delay path in a circuit. To overcome this weakness, timing verification programs such as Crystal [26], TV [27] and MOTIS [28] have been developed.

Model: The Lin-Mead (LM) delay model considers the delay of a falling (rising) edge of unit voltage to be the area under the $v(t)$ (or $1 - v(t)$ for the rising edge case) curve for the edge. This model gives an accurate yet simple expression for computing this delay in terms of node capacitances and link resistances. The delay computed takes into account effects such as dynamic switching of MOS capacitance and variable charging paths for a node. The nonlinearities of MOS transistors are ignored by the LM algorithm, which considers transistors to have a constant resistance when closed and nodes to have constant capacitance. The other major weakness of the simulation model is that the definition of delay does not model the detrimental effects of edge degradation due to long pass-transistor chains; however, experimental results in [23] indicate that the delays computed using this model compare quite well to delays arrived at using the SPICE circuit simulator [14].

Data Structures: To support delay calculation a resistance field, **RES**, must be added to each link record and a capacitance field, **CAP**, must be added to each node record. Also, accumulated charge, **CHRG**, cumulative capacitance, **CCAP**, cumulative delay, **CDLY**, and parent, **PNP**, fields must be added to each node record to be used in the delay calculation. Two new stacks for capacitance relaxation, **CSTK**, and delay calculation, **DSTK** are required to schedule these operations. Also, to implement event-driven logic update, the **LSTK** must be made into a list of stacks with a stack for each simulation time at which an event is scheduled.

Algorithm: The delay calculation algorithm is based on the TREE algorithm of [23] but has been significantly modified for compatibility with MOSSIM. This algorithm restricts itself to finding a single *source* which supplies current to each node. The *source* is found during the strength relaxations by setting the parent node pointer, **PNP**, field of each node to point back along the *dominant path*. While in general there may be more than one dominant path to a node, the algorithm finds only one path. This approximation is justified by the following facts: 1) for most simple cases gates, there is only one dominant path, and 2) in all cases, the algorithm gives a delay which is at least as long as if all dominant paths were considered.

The algorithm proceeds in two steps which occur between the *up/down relaxation* and *logic update* steps described above. During the *capacitance calculation* step, the capacitance seen by each node is computed.

This value is the sum of the node's capacitance and the capacitance of all other nodes to which it supplies current. These capacitance values are used during the *delay calculation* step in computing the transition time of each changing node. The algorithms for these two steps are listed below:

```
While CSTK[A] is not empty
    For each node on the CSTK[A], n
        If n has a parent
            m := node[n.PNP]
            m.CCAP := m.CCAP + n.CCAP
            schedule m on the CSTK[B]
        Else
            schedule n on the DSTK
    Swap CSTK[A] and CSTK[B]
```

Capacitance Calculation Algorithm

The capacitance calculation algorithm recursively computes the *dynamic* capacitance seen by each node in the *dominant path* of the nodes scheduled on the CSTK by working from the bottom of the path tree up. The initial scheduling of *leaf nodes* (nodes which are the *parent* of no other node) is performed during the strength relaxation step. The cumulative capacitance of nodes in a perturbed vicinity are initialized to the node capacitance during the perturbation step. During the first pass, each leaf node adds its capacitance to its parent's cumulative capacitance. On each subsequent iteration, each parent scheduled by the last iteration adds its capacitance to its parent's cumulative capacitance. The process terminates when a node with no parent is discovered. These nodes are the *source* nodes and are scheduled on the *DSTK* to be used in the delay calculation algorithm shown below:

```
For each node on the DSTK, n
    n.CDLY = 0
    For each link, l, in n.LP, with l.TS = 1 or l.TS = 2
        m := node[l.NP]
        if m.PNP = n then
            m.DLY = n.DLY + l.RES * n.CCAP
            schedule m on the LSTK
            if m is not a leaf
```

schedule m on DSTK

DELAY CALCULATION ALGORITHM

While the capacitance calculation algorithm works from the bottom of the path tree up, the delay calculation algorithm works from the top of the tree (the root) down. Delays are calculated recursively with each node computing its descendants' delays and then scheduling these descendants to compute their descendants' delays. The process terminates when the leaves of the tree are reached. As each node's delay is calculated, it is scheduled on the LSTK for logic update.

Timing Simulation: The delay calculation algorithms described above are designed to efficiently dovetail into the MOSSIM algorithm. The perturbation step is modified to reset the node capacitances to the initial capacitance modified by the accumulated charge, and the relaxation steps are modified to set the parent pointers and to schedule the leaf nodes on the CSTK.

The actual timing is implemented in the scheduling of logic updates. Each changing node is scheduled on the LSTK for update at the current simulation time plus the node's delay time. A separate stack in the LSTK set is maintained for each time during which updates are scheduled. During each logic update step, the simulation time is advanced to the next time for which an LSTK exists, and nodes to be updated are removed from this LSTK.

Note that a node which has been scheduled for update may change again before its update occurs. In this case the cumulative charge on the node must be recalculated. Using this value of charge, a new delay time is computed. The node is removed from the LSTK and rescheduled for update at its new delay time.

This phenomena models *glitches* which occur due to switching hazards. For example, consider the 2x2 and-or-invert gate shown in figure 3, below. If the transitions and delay times on the inputs are as shown, nodes a and b will be updated to be low at 10ns, turning the left leg of the gate off and scheduling the output node, e, to rise at 30ns. At 20ns, nodes c and d are updated to be high, turning the right leg of the gate on and rescheduling e to fall at 30ns. It is not clear whether the logic state of e should be set to X during this interval or whether it should remain zero.

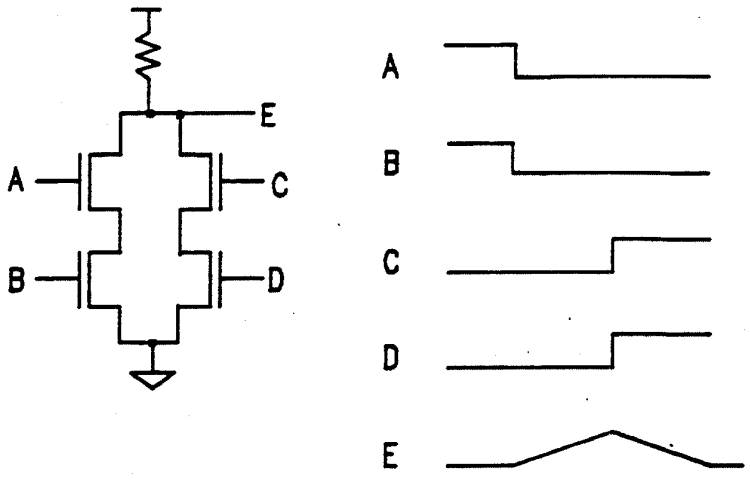


Figure 3. 2x2 And-Or-Invert Gate

Timing Verification: A timing verifier [26],[27] is concerned with calculating the longest delay time between clocked points. Such a verifier can be implemented using the delay algorithms described above by finding all paths between clocked points and propagating an edge over each of these paths. Such a timing verifier can be implemented on the MSE by replacing the simulation steps of the MOSSIM algorithm with path tracing steps.

Timing verification can be performed more efficiently by using a hierarchical circuit model [29]. This method characterizes the delay characteristics of a *leaf cell* in terms of its setup, hold, and propagation times. The delay characteristics of composition cells are arrived at by combining the delay characteristics of the leaf cells. Unfortunately, the architecture of the MSE restricts it to manipulating *flat* networks, making it unsuitable for hierarchical delay analysis.

3.4 Circuit Simulation:

Situations often arise in which timing simulation using simplified delay models such as those described above are not sufficiently accurate. For instance, in computing the delay of the bootstrap driver shown in figure 4 below, it is essential that the gate-channel capacitance of the bootstrap transistor be modeled. In these cases, circuit simulation is required. In circuit simulation we drop the abstraction of a signal as a logic level and consider the actual voltages on the nodes of the network.

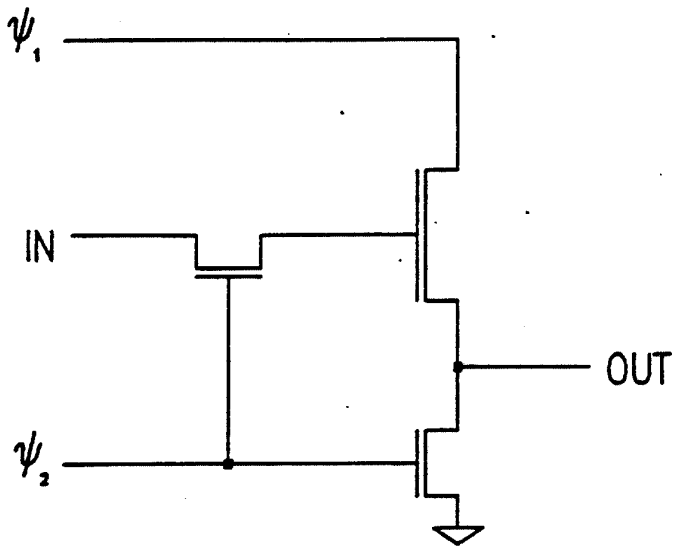


Figure 4. Bootstrap Driver

Circuit simulators vary from extremely accurate simulators such as SPICE [14] with very complete models to simulators such as MOTIS [13] which model only limited networks which are approximations of MOS logic circuits. Comprehensive circuit simulators such as SPICE are prohibitively time consuming for large circuits. Such simulators use detailed nonlinear device models and iteratively compute the circuit state at each instant of simulation time by means of sparse matrix techniques, Newton-Raphson iterations and stiffly stable, variable step and order, integration methods [30] [31].

Since the MSE is intended as a machine for the simulation of logic circuits, it is clear that a comprehensive simulation algorithm such as implemented in SPICE is not required. Also, the MSE provides no mechanism for manipulating sparse matrices making the implementation of a detailed circuit simulator on the MSE a difficult proposition. Another disadvantage of simulators based on the iterative solution of matrix equations is that unless the equations are decoupled, the inherent locality of activity in the circuit cannot be used to advantage. The SPLICE program [32] overcomes some of these problems by using mixed-mode simulation; however, subcircuits simulated in the timing mode still require the iterative solution of matrix equations. SPICE also takes some advantage of latency by using *bypassing* to avoid recomputing the values of circuit elements which are not changing. The circuit equations, however, must still be solved for all nodes.

An approximate circuit simulation algorithm such as used by MOTIS [13] lends itself to implementation on the MSE. MOTIS accurately simulates the nonlinear resistance of MOS transistors, and generates voltage waveforms assuming constant node capacitance. In order to decouple the circuit equations for each node, however, MOTIS ignores the effects of node-to-node capacitance such as found in bootstrap

circuits. In this section, we consider an adaptation of the MOTIS algorithm which is extended using waveform relaxation techniques [30] to simulate the effect of bootstrap capacitance.

Models In the network model for circuit simulation, the abstraction of logic state is dropped. Nodes take on discrete voltages and links take on discrete conductances depending on their gate source voltages. Each node has a constant node-to-substrate capacitance similar to the capacitance in the timing model. As in MOTIS, the nonlinearity of MOS transistors is implemented using a lookup table to avoid the overhead of computing transistor equations. Node-to-node capacitance is modeled using a special type of link which connects two nodes and is assigned a value of capacitance. To decouple the circuit as much as possible, internode capacitance will be ignored whenever the coupling capacitance is less than some fraction of both nodes node-to-substrate capacitances.

Algorithms The basic MOTIS algorithm [13] with no internode capacitance involves solving the following integral equation for the node voltages at each time step:

$$v_n(t_2) = v_n(t_1) + \frac{1}{C_n}\int_{t_1}^{t_2} i_n(t)dt$$

The current, i_n, is defined to be the current into node n and is calculated by summing the link currents into the node:

$$i_n(t) = \sum_m i_{mn}(t)$$

$$i_{mn}(t) = G_{mn}(t)(v_m(t) - v_n(t))$$

where G_{mn} is the conductance of the link connecting nodes m and n. For maximum stability, the backward Euler method is used to solve the integral equation for $v_n(t_2)$:

$$v_n(t_2) = v_n(t_1) + i_n(t_2)\delta t/C_n = v_n(t_1) + \frac{i_n(t_1)}{(\frac{C_n}{\delta t} + D)}$$

where D is the derivative:

$$D = \frac{di_n}{dv_n} = \sum_m G_{mn}$$

The solution of the backward Euler equation for node voltages is performed in an event driven manner so only nodes which are undergoing transitions are simulated.

When internode capacitances are considered, the algorithm described above provides an initial estimate of the node voltages, v_n. A relaxation is then performed to satisfy the charge requirements of the node-to-node capacitance. Consider the circuit of figure 5 below. Two nodes, n_i and n_j, with node-substrate capacitances C_i and C_j, are connected by a capacitor, C_{ij}. Neglecting the inter-node capacitance, the

algorithm above will compute the change in voltage on nodes i and j, $\Delta v_i = v_i(t_2) - v_i(t_1)$ and $\Delta v_j = v_j(t_2) - v_j(t_1)$. This gives an incremental voltage across the inter-node capacitor of $\Delta v_{ij} = \Delta v_i - \Delta v_j$. To cancel this voltage, a quantity of charge,

$$q_{ij} = \frac{\Delta v_{ij}}{\frac{1}{C_i} + \frac{1}{C_j} + \frac{1}{C_{ij}}}$$

must be transferred from C_i to one end of C_{ij} and from the other end of C_{ij} to C_j. The resulting change in voltage around the loop cancels the unsatisfied ΔV_{ij}. The new node voltages are given by $v_i' = v_i - q_{ij}/C_i$ and $v_j' = v_j + q_{ij}/C_j$. When several nodes in series are connected by inter-node capacitance, a single step solution is not possible and the waveform relaxation method is used to iteratively arrive at the solution.

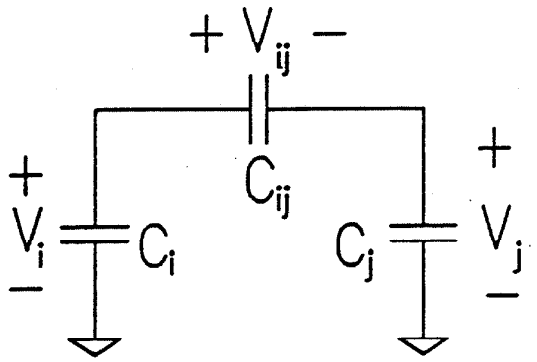


Figure 5. Inter-Node Capacitance

3.5 Functional Block Simulation:

Functional block simulation involves simulating a portion of a circuit with a procedural description. The four primary uses of functional block simulation in the development of a VLSI circuit are:

1. A functional block may be used to simulate the environment a circuit is to operate in. For instance, the memory and I/O devices seen by a microprocessor may be simulated by a functional block.

2. Portions of a circuit which are specified but not yet designed may be simulated by a functional block. This technique supports a top down approach to VLSI design where a chip is first described at the functional block level and then each functional block is designed and replaced by its circuit level description.

3. To suppress detail, a functional block may be used to simulate a portion of the circuit which has already been designed and functions correctly.

4. A functional block may be used to generate input vectors and test output vectors from a switch level simulation reducing the bottleneck associated with transferring vectors to and from a simulation.

In each of these cases, the functional block speeds simulation and increases the capacity of the simulation machine by summarizing the behavior of a large number of circuit elements in a single functional description.

For the MSE, auxiliary processors (APs) will support a functional simulation environment based on the DSIM functional simulator [24]. DSIM is an event-driven functional simulator which simulates interconnections of devices described in a procedural language. The timing primitives in DSIM can be used to model zero, unit, absolute, ambiguity or inertial delays providing for a great deal of simulation flexibility. Each device is compiled into procedures for:

1. initializing the device,

2. loading and storing the state of the device,

3. printing the status of the device,

4. evaluating the device.

The device procedures interact with kernel procedures for dealing with signals and events.

Data Structures

The DSIM data structure consists of a list of devices and a list of signals to represent the network, and an event list containing pending events.

Each signal has a name, dimension, and a type. A signal's name is an identifier used to symbolically refer to the signal. Internally all signals are referred to by pointers. A signal's dimension specifies how many bits a signal contains. There are two types of signals: root and derived. A root signal is a signal which is not made up of other signals. A derived signal, on the other hand, is a signal which is specified as the concatenation of subranges of other signals, root or derived.

A root signal also has a state, a pin list and a daughter list. Signal state has three attributes:

- **Values** a bit vector with length equal to the dimension of the signal. Each bit corresponds to the

value of one of the nodes the signal represents.

- **Valid:** a Boolean which is TRUE if the signal is valid, and FALSE if any bit of the signal is an X.
- **Drive:** a Boolean which has value TRUE if the signal has strength ω and FALSE if the signal has strength λ.

The pin list of a signal is a list of pointers to device ports which are connected to the signal and includes each ports contribution to the state of the signal. Each pin record also contains the time of the last update. This time attribute is necessary for implementing some delay models.

The daughter list of a root signal is a pointer to every derived signal which has a subrange of the root signal as one of its components.

In addition to name, dimension and type, derived signals also contain a list of parent records: pointers to root signals and the subrange of the root signal which is used.

A device record consists of a device name, a pointer to the device routines, a device type, and pointers to each signal connected to the devices ports.

The event list is a two level linked list. At the top level, *time records*, for each scheduled time are linked together in order. Each time record contains a time and a pointer to the root of a list of *event records* representing events scheduled to occur at the specified time. An event record contains a pointer to the device causing the event, a pointer to the signal (root or derived) on which the event is to occur, and the state to which the pin connected to this signal should be set.

Algorithms:

DSIM imposes no device models on the user, all device models are derived from the procedural description of a device. Because of this modeling flexibility, DSIM can even be used to simulate analog devices by considering signals to represent integer voltage levels rather than bit vectors.

DSIM does, however, provide a model for a signal and how signals interact. When a device wishes to set a signal it calls the DSIM event scheduling procedure. To examine a signal a device calls the DSIM getValue procedure, and when a scheduled event occurs, DSIM uses the circuit connectivity and device sensitivity information to assure that all affected devices are re-evaluated.

Event Scheduling: To schedule an event, a new event record is initialized and inserted at the end of the event list for the scheduled time. If this is the first event to be scheduled for a particular time a new time record is created.

Event Processing: When all events at the current simulation time have been evaluated, the simulation time is advanced to the next time record in the time list, and the events pointed to by that time record are evaluated. Each event is removed from the event queue. If the event occurs on a derived signal it is decomposed into root events and each root event is processed separately.

For each root event the first step is to find the new state of the root signal. If there is no change of state no further processing is performed. If the state did change, every *sensitized* pin connected to a subrange of the signal which changed is evaluated (possibly scheduling new events).

For performance enhancement, DSIM uses a *sensitization* mechanisms similar to that implemented in the ADLIB-SABLE system [33]. By providing a sensitization mechanism, a block may control which of its inputs are *sensitized* to transitions. Only when a sensitized input changes state will the block be evaluated.

Interfaces

To interface DSIM signals to MOSSIM nodes, we map each bit of a signal to one node, and use the drive of a signal to determine the strength of all the nodes it reduces to. With this model, bidirectional ports which can be either inputs or outputs can be implemented. When in the input state, drive = FALSE, these nodes are assigned size κ_1 so that their logic state will be controlled by the switch-level simulation. In the output state, the node strengths are changed to ω so output nodes are treated as inputs to the switch-level model.

Chapter 4

Concurrency

To achieve the performance required of a hardware simulation engine, concurrency is required. In this chapter we examine the MOSSIM algorithm presented in sections 3.1-3.2 for potential concurrency and propose techniques for improving the performance of this algorithm in hardware. These techniques are used as the basis of the architecture described in chapter 5.

The philosophy of the MSE is to optimize the performance of MOSSIM while providing *hooks* for other levels of simulation. Thus, we are not concerned with optimizing the performance of the circuit or timing simulation algorithms discussed in chapter 3, and we will restrict our attention to the MOSSIM algorithms in this chapter.

4.1 Functional Concurrency

To analyze the potential functional concurrency of the MOSSIM simulation algorithm, we attempt to partition the algorithm into *separable* functions. For two functions to be separable, they must not access the same data structures, and they must not *block* each other so they can proceed concurrently. Three functions are used across the four simulation steps which comprise the MOSSIM algorithm. They are:

1. *Node Scheduling* involves the manipulation of the event stacks and implements two operations: 1) *scheduling* nodes on the event stacks, and 2) *retrieving* a previously scheduled node. Nodes are scheduled for operation by pushing a pointer to the node on the appropriate event stack (possibly as a function of its strength). In the algorithm, this operation is indicated as: **schedule <node> on <stack>**. Scheduled nodes are retrieved from the event stacks by popping the pointer to the node off the appropriate stack. The statement, **For each node, <node>, on <stack>**, in the algorithm represents this operation.

Node scheduling meets the criteria for a *separable function*. The two operations of *scheduling* and *retrieving* nodes are the only operations which access the event stacks, and these operations may proceed in parallel with the other operations in the algorithm. The *schedule* operation is initiated by another function, but that function need not wait for the operation to complete. Thus, the *schedule* operation and the initiating function may proceed in parallel. The *retrieve* operation on the other hand is the outermost loop of each algorithm and acts as the initiating function. Since it need not wait for the function it initiates to complete before retrieving the next node, the two may execute concurrently.

2. *Network Traversal* represents the second nesting level in each algorithm. The statements **For each link, l, ...** and **For each gate record, g, ...** represent invocations of the network traversal function. Network traversal is not a *separable function* in the strict sense since the link list is accessed by other operations in the logic update step. However, in all other simulation steps, this function is separable and concurrency can be exploited by making it a separate process. Also, the operations of link list traversal and gate list traversal are both *separable functions* in their own right. However, it would be a waste of resources to implement the gate list traversal function separately as it is only used in the logic update step of the MOSSIM algorithm, and (as is shown in chapter 7) only 10%of simulation time is spent in the logic update step.

3. *Node Operations* include all updating of the node records in the relaxation and perturbation algorithms. These are the only operations which access the node list and may proceed concurrently with both scheduling and traversal.

By partitioning the MOSSIM algorithm into three function units as shown in figure 6 below, a potential threefold improvement in performance is possible. However, for this performance improvement to be realized, the execution of the three functions must be balanced so that one function does not dominate as the rate limiting step.

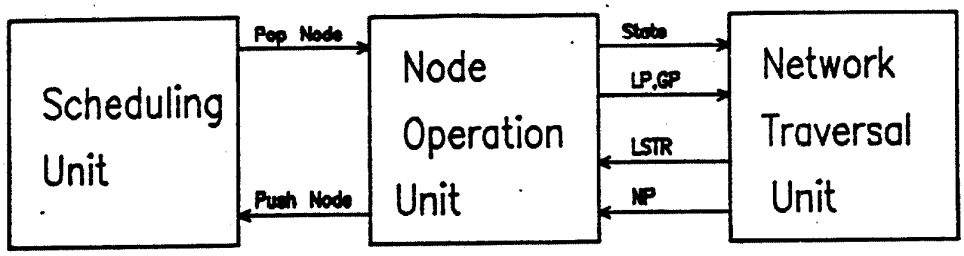


Figure 6. Functional Partitioning of the MSE

An important observation is that each of the separable function units is associated with one (or two in the case of the network traversal function) of the main data structures described in section 3.1. In fact, if we separated the link and gate traversal functions, there would be a function unit for each data structure. This observation leads us to a general method of finding functional concurrency in special purpose algorithms by grouping each data structure with the operations which can be performed on it, into a function unit.

The question is at what level to break the system into function units. For instance, the node list could be split into two function units with half the nodes in each unit. Such a split would be an example of *subcircuit concurrency* applied within a unit of *functional concurrency.* To determine the level at which to partition the system and to decide whether to nest *subcircuit partitions* within *functional partitions* or vice-versa, we must consider the issues of *balance, utilization* and *traffic.* The units must be *balanced* so that no one unit or collection of units dominates the execution time. Enough potential concurrency between units must exist so their *utilization* is high. If two units are rarely in use at the same time, their *cross utilization* is low, and they may be combined into a single unit with very little degradation in performance. The final partitioning issue is *traffic.* When a function is partitioned into two *units* which communicate with each other, we actually create three units: the two function units and the communications channel between them. The message *traffic* between the units determines the cost of this channel and thus affects the feasibility of a partition. Section 4.2 discusses how specialized circuitry can be added to balance the capability of the function untis and to keep their utilization high. The issue of nesting is addressed in section 4.3 after we have taken a look at subcircuit concurrency.

4.2 Specialization

Just as specialized logic is used in the YSE (section 2.1) to evaluate the output of a four-input logic gate, specialized logic can be applied to the relaxation, logic state determination and perturbation operations of the MOSSIM algorithm. Consider for example, the node operation function in the up/down relaxation algorithm. Thirty operations such as field-extract, field-insert, min, max and compare are required to compute the new up and down strengths and logic state of node **n**. Implemented in a conventional programming language such as Mainsail [34] this sequence would require upwards of 100μs to execute on a VAX 11/780. By implementing this function in special hardware, the entire operation can be performed in 200ns, a 500:1 improvement.

If specialization is applied to the Node Operation function, then specialization must be applied to both the Scheduling function and the Network traversal function to achieve balance. The following graph illustrates a typical MSE execution sequence. We can expect in general four links traversed, one source operation and two destination operations and one push for each node popped off the scheduling unit stack.

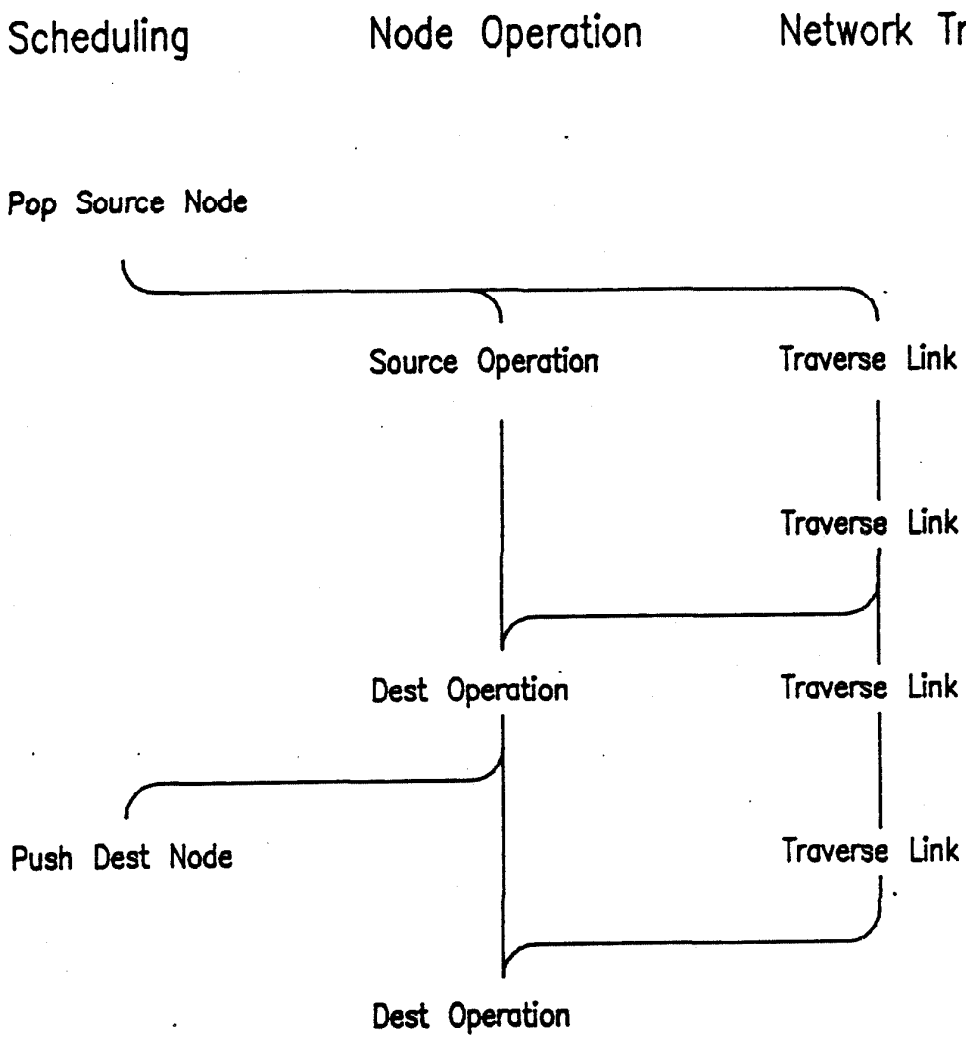


Figure 7. Typical MSE Execution Sequence

To achieve 100%utilization we would have to make a traversal operation take 0.75 as long as a relaxation operation, and a pop take twice as long as a relaxation operation. This scheduling is shown in figure 8, below. Fractional timing and the level of pipelining required to achieve the optimal utilization, however, make 100%utilization impractical.

The schedule implemented in the MSE, shown in figure 9, allows one time unit (200ns) for relaxation, push and traversal and two time units for pop. This scheduling achieves 80%utilization and is simple enough to be implemented with two levels of pop pipelining, one level of traversal pipelining and four levels of relaxation pipelining.

SU	NOU	NTU
Pop	Source Op	Traverse
		Traverse
	Dest Op	Traverse
Push	Dest Op	Traverse

Figure 8. Optimal Utilization Schedules

SU	NOU	NTU
Pop	Source Op	Traverse
		Traverse
	Dest Op	Traverse
Push	Dest Op	Traverse

Figure 9. Actual Utilization Schedules

4.3 Subcircuit Concurrency

We have already touched on the issue of subcircuit concurrency in section 4.1 by considering a partition of the node list. Such a partition is in fact motivated by the locality of activity inherent in switch-level simulation. In each of the algorithms, activity begins at a node (by retrieving this node from a stack), and spreads outward one link at a time (by scheduling the *neighbors* of the node for evaluation). It would seem that if two nodes in different parts of the network were scheduled on a stack, processing of these nodes could proceed in parallel with little interference. To test this conjecture we must consider the *locality* of MOS networks.

Locality: In order to determine the potential for subcircuit concurrency in the MSE, we have made a study of locality in MOS transistor networks [35]. In general, the *locality of a partition of a directed* graph is the fraction of arcs which begin on a vertex in the partition that also end on a vertex in the partition. A partition of an MOS transistor network exhibits two types of locality. The graph defined by the link list of the network defines its *connection locality*, and the graph defined by the gate list of the network defines the network's *gate locality*. Figure 10 below shows the mapping from an MOS transistor to edges in the link and gate graphs. For each transistor with source, drain, and gate nodes, n_s, n_d and n_g respectively, arcs exist in the link graph from n_s to n_d, and from n_d to n_s. Arcs exist in the gate graph from n_g to both n_s and n_d.

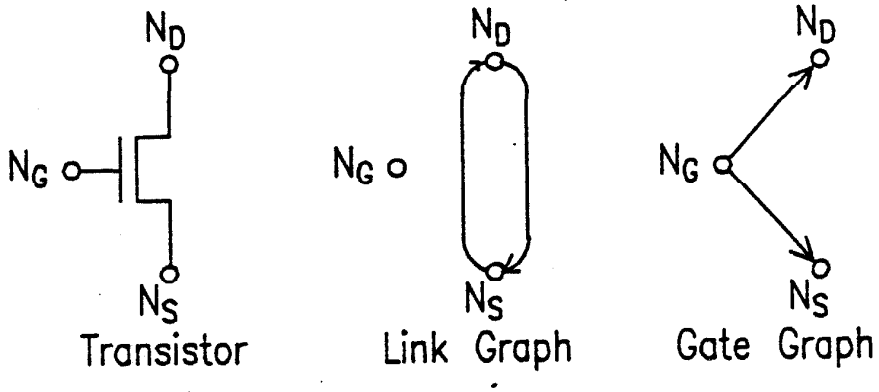


Figure 10. Mapping from Transistor to Link and Gate Graphs

Using this definition of *locality*, we have made locality measurements on several VLSI circuits. The results of locality measurements on the MOSAIC chip [36] are shown in figure 11 below and summarized in the following table. The MOSAIC chip is a 16-bit microprocessor and is composed of 9239 transistors

and 3452 nodes. The partitioning was performed using a heuristic algorithm described in more detail in chapter 7.

Partition Size	Number of Partitions	Connection Locality	Gate Locality
32	120	91.0%	47.8%
64	59	92.3%	60.5%
128	29	93.1%	68.6%
256	15	93.8%	73.5%
512	8	96.1%	75.9%
1024	4	100.0%	78.5%
2048	2	100.0%	94.4%
4096	1	100.0%	100.0%

Table 16. Locality of the MOSAIC Chip

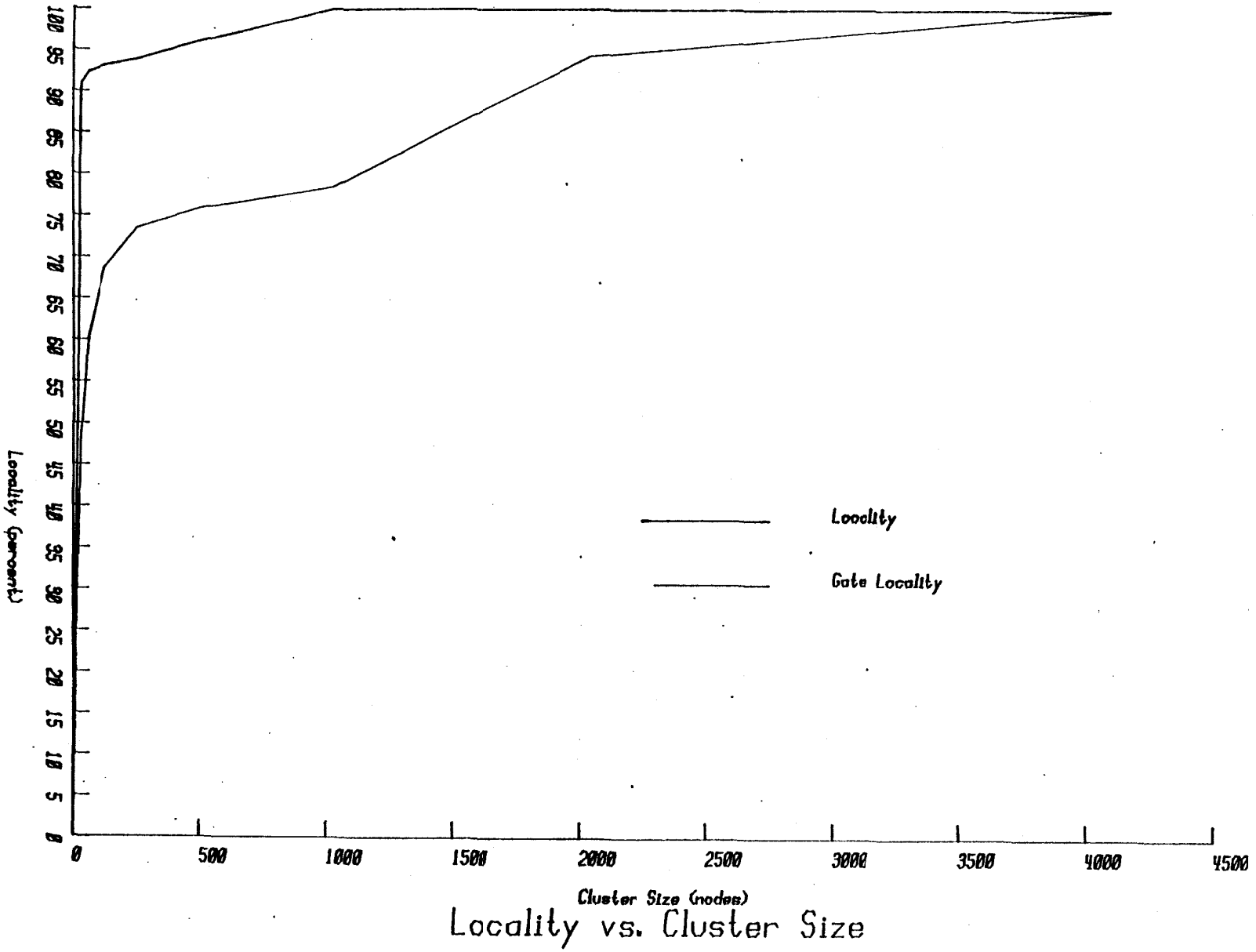


Figure 11. Locality of the MOSAIC Chip

These locality figures indicate that MOS transistor networks have a great deal of locality, especially connection locality. With partitions as small as 32 nodes, 91%of the links are local to a partition and 48%of the gates are local to a partition. The high connection locality can be understood by examining the *node groups* of the MOSAIC chip. A node group is a set of all nodes which belong to a connected subgraph of the network's link graph. The 1740 node groups of the MOSAIC chip range in size from 1 node to 976 nodes. As can be seen from the histogram of figure 12, most groups contain fewer than 5 nodes. In fact the mean group size is 4.06 nodes (with a standard deviation of 33.08 nodes) and the median group size is 3 nodes. Thus, for partitions of greater than 4 nodes, most links would be local to the partition and the connection locality would be greater than 50%.

The 'spikey' nature of the frequency distribution occurs because of the regularity of the chip. Common subcircuits, each having the same group size distribution, account for the spikes. For example, an inverter contains one group of size 1, a two input NAND gate contains one group of size 2, and so on. Circuits such as an ALU bit-slice may have several groups, and when 16 of these circuits are repeated to form an ALU, a spike of height 16 occurs in the frequency distribution for each of these constituent group sizes.

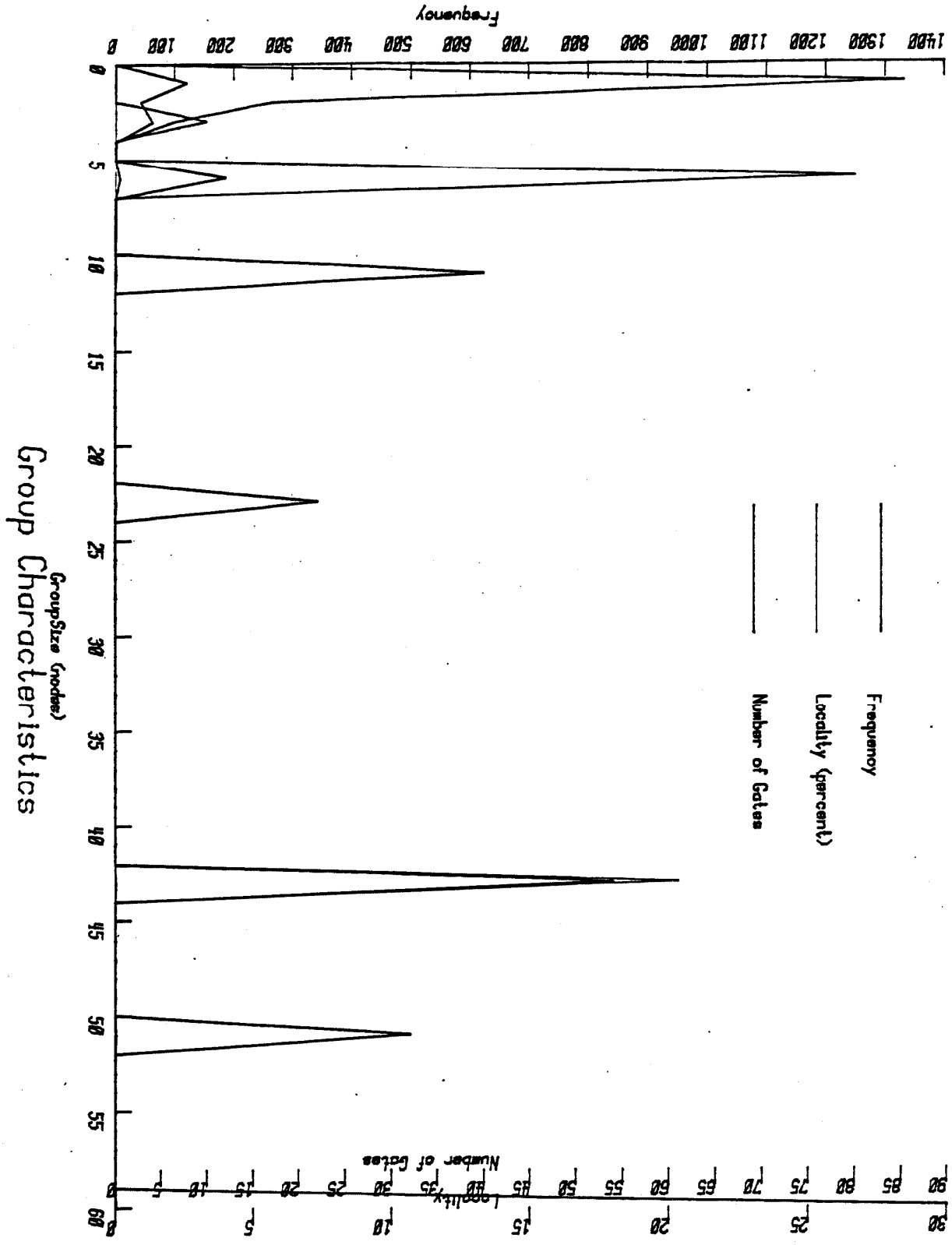


Figure 12. Frequency of Node Group Size in the MOSAIC Chip

The gate graph exhibits no natural partitioning similar to a *node group*. From the data of table 16 it appears that gate locality, l_g, as a function of partition size, N_p, and network size, N, follows a power function:

$$l_g = \left(\frac{N_p}{N}\right)^{1/3}$$

This function is similar in form to Rent's rule [37, 38], but differs significantly both in definition and in the value of the exponent. The major reason for this variation is the severe limit on the number of off chip connections forced by the small pinout of an integrated circuit package.

Up to now we have been discussing *static* locality. Since gate connections are always present, the concept of *gate locality* is static in nature. Links, however, can dynamically change state, varying the connectivity of the link graph. If we define the *X-closed link graph* as the link graph with all *open* links removed, and the *closed link graph* as the *X-closed link graph* with all *unknown* links removed, then these new graphs define two types of *dynamic* connection locality. It is this *dynamic* connection locality which affects the performance of the MOSSIM algorithm. In fact, *node groups* on the dynamic link graphs correspond exactly to the *vicinities* discussed in section 3.1 [10].

We believe that MOS transistor networks have much higher *dynamic* connection locality than *static* connection locality. Consider for example the 976 node group of the MOSAIC chip. An analysis of this group revealed that it contained a majority of the nodes in the data path of the chip, including each bit of the data bus. In operation, the data bus lines would never be shorted; however, a shifter network provides *potential* connections between these lines. Since a static analysis does not consider the mutual exclusion of connections inherent in the shifter, it concludes that the busses are connected. The largest *vicinity* in the operating data path is less than 20 nodes. In this case at least, the dynamic locality is far greater than the static locality.

To determine the degree of dynamic locality, the MSE functional simulator has been instrumented to measure a third type of dynamic locality, *active* connection locality. This type of locality considers only those links over which relaxations occur during one cycle of the MOSSIM algorithm. *Active* connection locality differs from the dynamic localities described above in that it excludes stable *vicinities* from consideration. Unfortunately, the only circuits which could be simulated on the functional simulator were too small to give meaningful locality statistics. Once the prototype MSE is constructed, dynamic locality experiments on large circuits will be possible.

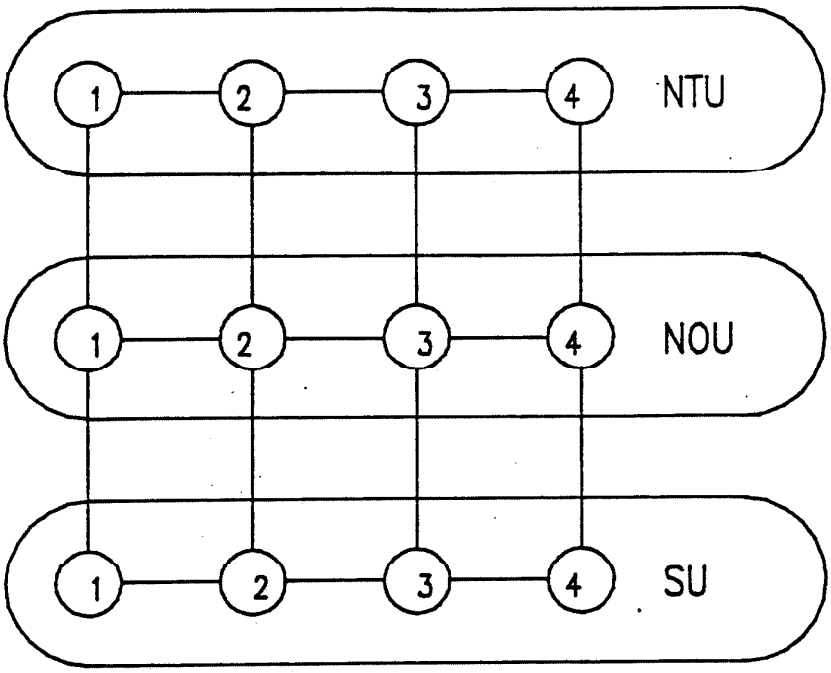
One locality statistic arrived at from the functional simulation of the MSE is the percentage of activity in a logic circuit. These activity statistics are presented in the following table. The table shows that as circuits grow in size the percentage of active logic decreases. Also, the style of the design affects the

activity level. A highly precharged design such as the Manchester counter has much higher activity than a comparable counter designed with a carry circuit built from gates. In all circuits, the activity is low in the range of 5%to 10%for all but the smallest circuits. Counters are not representative of all integrated circuits, however they do model the behavior of adders and ALUs. Other common components of VLSI chips: RAMs, ROMs and PLAs exhibit even higher locality as only a few word lines (usually 1) out of many change state during any one test vector.

TRANSISTORS	ACTIVITY	CIRCUIT
52	13%	4-BIT COUNTER (GATES)
320	4%	16-BIT COUNTER (GATES)
464	9%	16-BIT COUNTER (MANCHESTER)

Table 17. Percentage of Activity in Counters

Nesting: The locality results described above strongly suggest nesting functional concurrency within subcircuit concurrency. To illustrate this point, consider the two possibilities shown in figures 13 and 14 below.



SF Nesting

Figure 18. SF Nesting

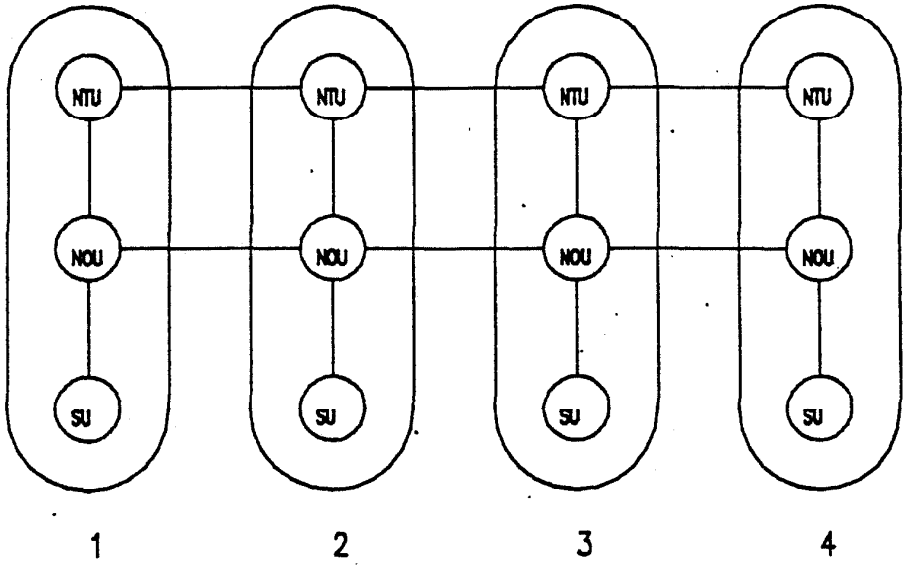


Figure 14. FS Nesting

In comparing these block diagrams we will consider that communication at the higher level of nesting is ten times as expensive as communication at the lower level of nesting. This cost function is motivated by the well known rule of thumb which states that a connection at each level of electronic packaging (IC, Board, Backplane, Rack, ...) costs an order of magnitude more than a connection at the level immediately below. For the subcircuit within functional (SF) nesting case, figure 13, all messages between function units must travel at the higher level of nesting giving a cost of $10M$ where M is the number of messages. In the functional within subcircuit (FS) case however, figure 14. local messages can travel at the less expensive lower level giving a cost of $(10 - 9l)M$ where l is a composite locality function. If we define $l = .75l_c + .25l_g$, then for the case of 32 node partitions, $l = 80\%$, and the FS message cost is only 28%that of the SF cost. The relative costs diverge even further for larger partition sizes.

Mechanisms As shown in figure 14, with FS nesting, the *network traversal* (NT) function units dominate the communication with other partitions. A hierarchical addressing scheme is used to implement this

communication. A hierarchical address is implied by the pointer fields of link and gate records. A normal or *internal* link (gate) contains a *local address* of the node (link) it points to. External link and gate records must append a *processor address* to the *local address* to give the complete *global address* of the node or link. Further, the global address carried by an external link (gate) is not a node (link) address, but the pointer to the beginning of a link (gate) list. Thus, several links from one node to a single external processor require only one external link record and only one message. If the subcircuit hierarchy is not limited to two levels then the hierarchical addressing scheme need not end at *processor address*. Any number of levels may be supported.

The gate and link type fields **LTYP, GTYP** described in section 3.1 indicate the level of addressing. A local address consists of a single link record with **LTYP** = *internal*, and a local node pointer in the **NP** field. External links are identified by **LTYP** = *external*. These links have the local address of the external link list within *its* processor in the NP field and are followed in the link list by a number of extension records with format:

LTYP	PP

The **LTYP** field is analogous to the same field in the link record indicating whether the extension address is local at its level of the hierarchy or if another extension record follows. The **PP** field is the next level address. For two level addressing, the **LTYP** field of an extension will always be either *internal* or *last*. (The *last* type represents an internal address which is the end of a node's link list.) The **PP** field will be the processor address for the two level case. This hierarchical addressing mechanism provides the MSE with unlimited extensibility in terms of network size.

4.4 Virtual Processors

The very locality of activity which makes subcircuit concurrency possible can lead to a degradation in performance due to idle processors. If the amount of activity in each processor is not equal, processors with low activity will complete a processing step early and remain idle until all processors complete the step. To avoid this potential degradation, we have developed the concept of *virtual processors*. This concept is analogous to that of *virtual memory* [39]. We partition the circuit into many more subcircuits than we have physical processors. A *virtual processor*, associated with each subcircuit, contains the complete state of the simulation of that circuit. To maximize throughput, the *virtual processors* are dynamically mapped to *physical processors* based on activity. As soon as a virtual processor becomes idle, it is *swapped out* to be replaced by a processor with activity. This mechanism minimizes the amount of time a physical processor is idle.

In order to implement the virtual processor concept, a swapping mechanism, mapping mechanism, and scheduling algorithm are required. In addition, a memory hierarchy is required to keep the state of swapped out processors. Each of these issues is addressed in the following paragraphs. In this discussion we use the terms *process* and *virtual processor* interchangeably.

Swapping is implemented at the process level by copying the entire state of a subnetwork process into backing store. If we restrict swapping to occur at the completion of the outer loop of each algorithm, none of the physical processor's working registers need be saved or restored. Only the node list, link list, gate list and event stacks (including stack pointers) need be copied. After the old process is swapped out, the new process is swapped in by copying its state from backing store.

Mapping is required on all interprocessor communications. An individual process uses only local addresses and requires no mapping. In fact, as long as it is not moved to a different processor group, it need not know which physical processor it is executing on. Mapping is implemented in the interprocessor message switch. All messages are transmitted with virtual addresses. The message switch queues arriving messages according to virtual address. A separate message queue is maintained for each process. A routing table in the message switch holds the current process to physical processor mapping and is used to direct output from the queues.

A Scheduling Algorithm controls the assignment of processes to physical processors. An efficient scheduling algorithm should optimize throughput by keeping all physical processors busy all the time. A candidate scheduling algorithm is shown below:

```
Starting From an Initial Assignment
While some process is not done
    If a process, p1, on processor P is done
        Select the swapped out process, p2, with the longest queue
        Swap p1 out of processor P to backing store
        Swap p2 in from backing store to processor P
```

VIRTUAL PROCESSOR SCHEDULING ALGORITHM

Memory Hierarchy: Because disk access times (order of 10ms) are much longer than relaxation times (200ns/relaxation), it is necessary to implement a hierarchical backing store. A three level hierarchical store is proposed for process swapping on the MSE. The three levels are: Disk, global RAM, and local

RAM. Disk accesses are buffered by global semiconductor read/write memory (RAM) large enough to prevent a physical processor from blocking for a disk access. Processes with high priority (long queues and full event stacks) are prefetched into the buffer in anticipation of an idle processor, and write-back of a swapped out process is buffered to avoid thrashing. To prevent a processor from sitting idle while a swap is performed over a link with the global RAM, a local RAM is provided in each processor with enough room for two processes. A high priority process can be stored in local RAM and swapped in quickly as soon as a processor idles.

Note: While the local RAM is still a part of the MSE architecture, it has been omitted from the prototype MSE due to the limited space on the prototype wire-wrap board.

Performance: To estimate the performance of the virtual processor mechanism we use a model which relates number of logic evaluations per swap, and swapping time to the size of a subnetwork, N. Two models are considered based on different estimates of locality of activity.

Model A:

Since the level of activity in a circuit is from 5%to 10%we estimate that T/10 logic evaluations will occur before a subnetwork is swapped where T is the number of transistors. Typically T $\approx$ 3N, so $\approx$ N/3 logic evaluations will take place between swaps. At a rate of 250K logic evaluations per second, $\approx$1.3N μs will elapse between swaps.

A swap to local RAM requires 200ns per 32-bit word. For a subnetwork with N nodes and 3N transistors, N words must be swapped out and 14N words must be swapped in. These 15N transfers require $\approx$ 3N μs to perform. (Network structure is not swapped out).

Model A assumes the activity is spread evenly throughout all subnetworks. As a result, no one subnetwork stays swapped in for very long and more than twice as much time is spent swapping subnetworks than is spent simulating.

Model B:

In this model we assume that the activity of a circuit is concentrated in a few subnetworks. The level of activity in these active subnetworks is very high. Since these subnetworks may change the states of some of their transistors several times before they are swapped out, we assume N logic evaluations are performed accounting for 4 N μs between swaps.

Since the swapping time remains as before, even with this optimistic outlook, about half the processor's time is spent swapping circuits. We plan to perform experiments on the prototype MSE to determine

what fraction of time the processor spends swapping. While the models described above predict a constant fraction for all sizes of subcircuit, intuitively one would expect the fraction to decrease as subnetwork size is increased. Also, this fraction should be highly dependent on the ratio of the number of physical processors to the number of virtual processors. Perhaps there is some *working set* [39] of virtual processors for which swapping becomes very low.

4.5 Conclusion

In this chapter we have examined the issue of concurrency in the MSE. A partitioning of the algorithm has been developed where three concurrently executing function units comprise one subnetwork processor, and several subnetwork processors executing concurrently comprise the MSE. Also, a virtual processor mechanism has been proposed to balance the load of the subnetwork processors. In the next chapter a hardware architecture is developed based on these ideas.

Chapter 5

Architecture

In chapter 4 we examined the concurrency which exists in the MOSSIM algorithm. In this chapter the special purpose hardware required to exploit this concurrency is described at the block diagram level. Design details are presented in chapter 6.

As shown in figure 15 below, the MOSSIM Simulation Engine (MSE) consists of a number of *subnetwork processors* (SP) connected by a message bus (MB) to a *message switch* (MS). Auxiliary processors (AP) may also be connected to the MB to perform functional simulation. A host processor (HP) controls the operation of the MSE and has the ability to read and write each register and memory location in the machine.

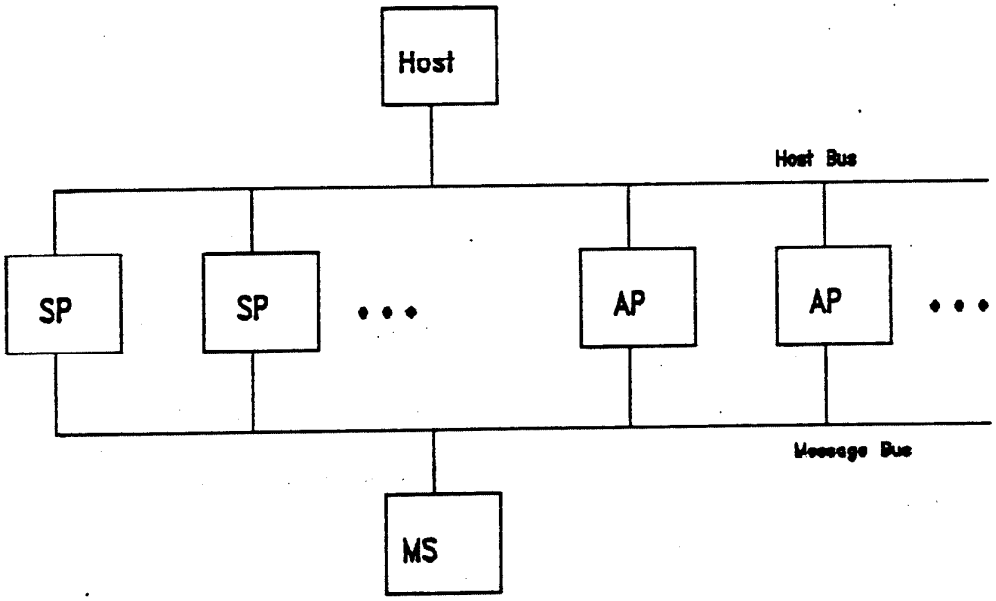


Figure 15. MSE Block Diagram

5.1 Subnetwork Processor (SP)

The SP is the hardware kernel of the MSE. It executes the MOSSIM algorithm described in section 3.1 performing all operations for its subnetwork, and sending messages to the MS for operations involving the rest of the network. Figure 16 below shows a block diagram of the SP. The three functional units described in chapter 4: the *node operation unit* (NOU), *scheduling unit* (SU), and *network traversal unit* (NTU) are augmented by an *I/O Unit* (IOU), *control processor* (CP) and the local swapping RAM (LSR).

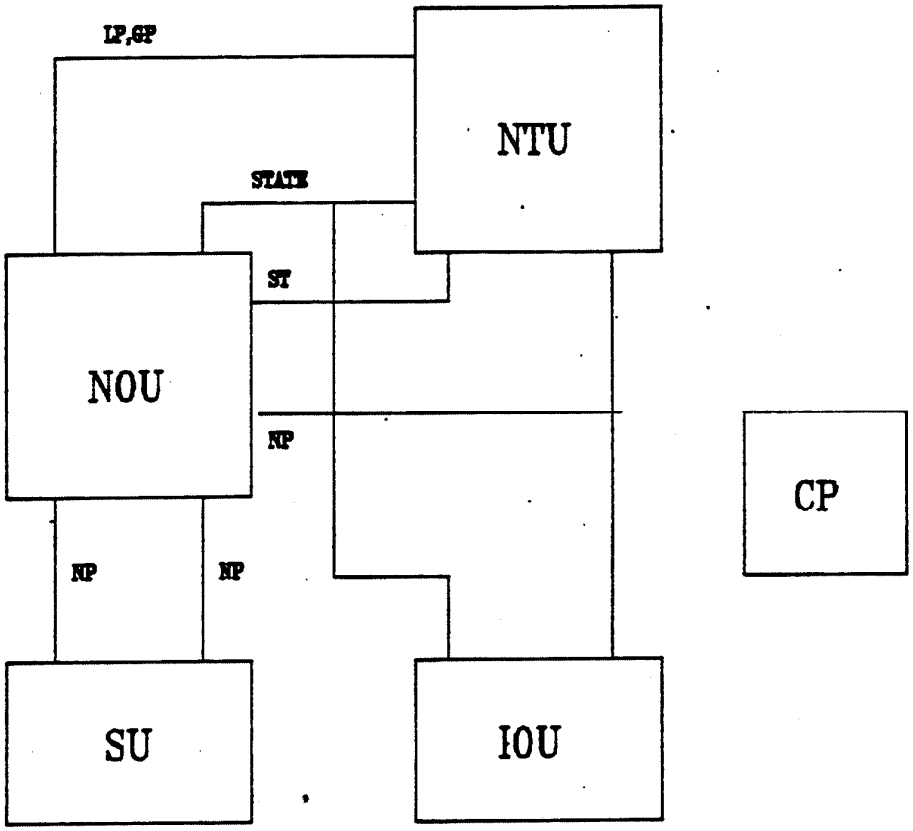


Figure 16. Subnetwork Processor Block Diagram

Control Processor: The CP is a microcoded processor. Its microcode directly controls the NOU and synchronizes the operation of the other three units. The CP data path has access to all the the SP memories. The functions of the CP are:

1. To control operation of the NOU, NTU and SU pipeline.
2. To perform swapping between SP memories and the LSR. (LSR not implemented in prototype).
3. To perform the arithmetic operations required by the timing and circuit simulation algorithms presented in chapter 3.

Node Operation Unit (NOU). The NOU, shown in figure 17 below, consists of the node list memory (NLM), four pointer registers, and the relaxation data path. Four registers and an relaxation unit (RU) comprise the relaxation data path. The source and destination node registers (SNR) (DNR) contain the state fields (no pointers) of the nodes at either end of a link being relaxed across. The relaxation result register (RRR) holds the result of the last relaxation operation performed by the RU. The net list memory data bus, NLMD, also connects to the NTU where two counter registers latch the LP and GP fields of the current source node, and to the IOU to transmit state information to the MS when relaxing against an external node. The relaxation unit contains special hardware to update the flag, strength and logic state fields of a node record.

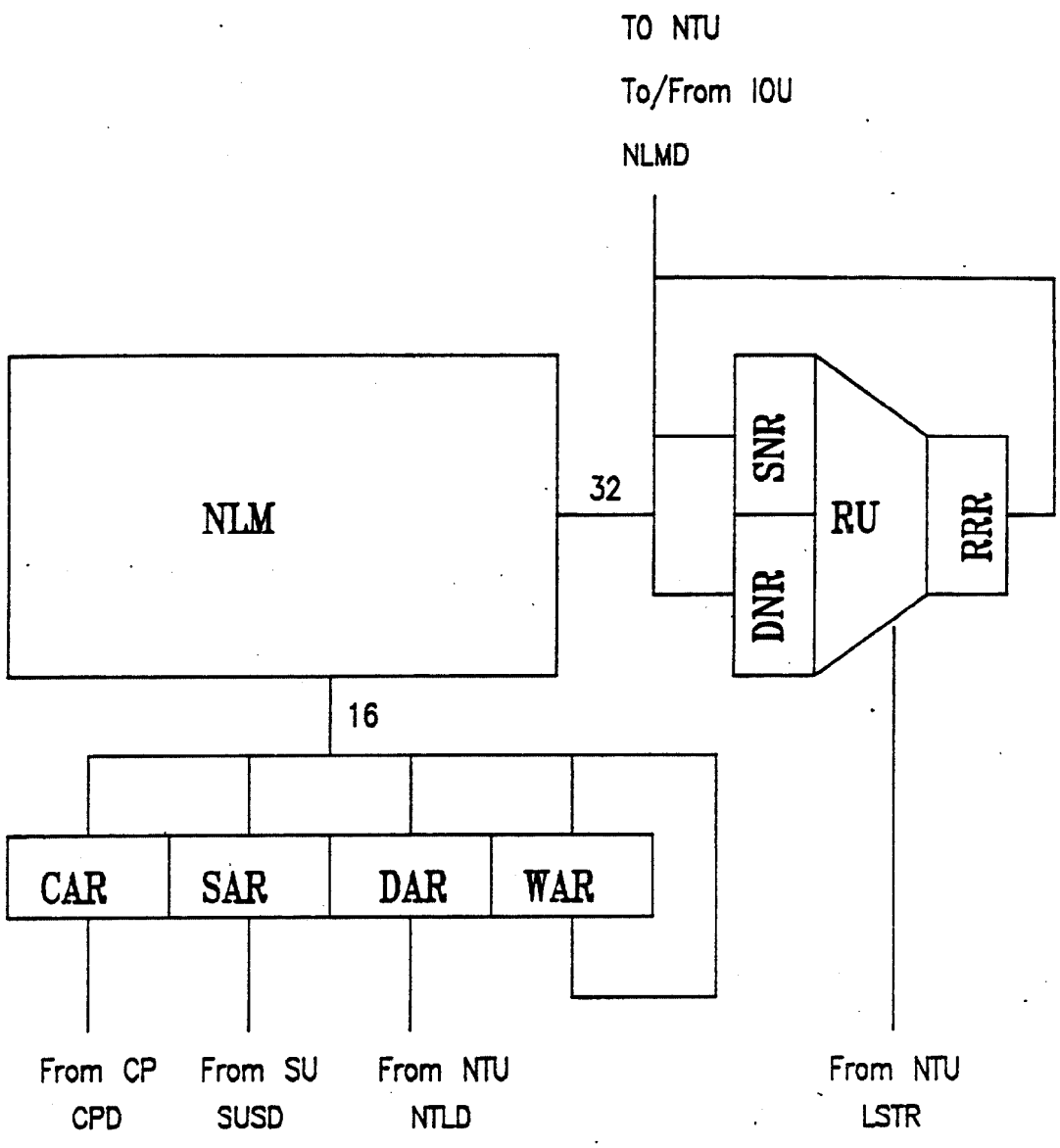


Figure 17. Node Operation Unit Block Diagram

During execution of a relaxation step, operation is as follows:

1. SAR ← Pop(SU).

2. NTU counters ← NLM[SAR+1],
   Start(NTU).

(Note, all node addresses are even).

3. SNR ← NLM[SAR],
LSTR, DAR ← next(NTU),
(NOU stalls until the NTU finds an active link or the end of the link list).

4. DNR ← NLM[DAR],
RRR ← resetFlagOp(SNR),
WAR ← DAR,
DAR ← next(NTU).

5. DNR ← NLM[DAR],
NLM[SAR] ← RRR,
RRR ← destOp(SNR,DNR).
WAR ← DAR,
DAR ← next(NTU),
If update1, schedule(SU, nextStep, SAR).
(Note, updates are set by the operation of the previous step).

6. DNR ← NLM[DAR],
NLM[WAR] ← RRR,
RRR ← destOp(SNR,DNR).
WAR ← DAR,
DAR ← next(NTU).
If update2, schedule(SU, thisStep, WAR),
If update3, schedule(SU, thisStep, WAR),

7. Repeat step 6 until the NTU finishes traversal and signals done.

Steps 1-5 perform the pipeline initialization and the activity flag resetting of the source node. First the address of the next node scheduled for this step is popped from the highest priority scheduling unit stack and loaded into the source address register (SAR). This operation actually consumes no time as the SU

keeps one node ahead of the NOU on pops. The SAR is simply loaded from the pop-ahead buffer in the SU. If the SU buffer is empty the NOU stalls until the SU performs a pop.

On the second step, the pointer portion of the source node is read from the NLM and loaded into the NTU traversal counters. The NTU is given a start command to begin traversal.

On step 3, the state portion of the source node record is read from the NLM and loaded into the source node register SNR. Also during step 3 the destination address register and the link strength are loaded from the NTU. If the NTU is not ready with an active link, the NOU stalls executing wait states until the NTU is either ready or done.

The current activity flag of the source node is reset during step 4, and the first destination node is read from the NLM and loaded into the DNR. The write-back address register (WAR) is loaded with the address of the destination node so it can be written back after relaxation. If the source node sets the update1 flag, it is scheduled for the next simulation step.

In step 5 the first actual relaxation is performed on the SNR and DNR with the result latched in the RRR at the end of the cycle. The RRR which contains the updated source node from step 4 is written back to the NLM at the address specified in the SAR. The next destination node (to be used in step 6) is loaded into the DNR, and the write-back address register(WAR) is loaded with the address of this node (at the end of the cycle). Again, the DAR is loaded from the NTU with a stall occurring if the NTU is not ready. Depending on the state of the update2 and update3 flags, the destination node (address in WAR) is scheduled for the current simulation step and/or the next simulation step.

The pipeline reaches steady state at step 6. There are four levels to this pipeline.

- Every cycle the address of the destination node to be relaxed two cycles from the present cycle is loaded into the DAR.
- The destination node to be relaxed on the next cycle is loaded from the NLM.
- The current destination node is relaxed against the source node.
- Finally the destination node relaxed on the previous cycle is written back to the NLM.

This extensive pipelining allows the MSE to perform one relaxation every clock cycle as long as the NTU and SU do not cause *stalls* by failing to supply it with destination and source node addresses. An elaborate control mechanism is required to orchestrate this heavily interlocked pipeline. Some features of the control logic are described in chapter 6.

Relaxation Unit: Figure 18 below shows a block diagram of the relaxation unit (RU). The RU operates on the values contained in the SNR and DNR to produce a result in the RRR. A special function unit for each field of the node record computes the next state of that field depending on the operation and the simulation step (one of the four listed in section 3.1). These steps will be referred to by their corresponding activity flag names R, U, L and P (see table 2). There are two operations used in the sequence shown above. In step 4, the resetFlagOp operation resets the activity flag of the source node record to indicate that it has been retrieved from the SU. This operation also sets the activity flag for the next operation if the update1 condition (see table 11) is met. The flag unit is always involved in this operation. During the logic update step the logic state unit is also involved as it copies the NLS field to the LS field. During the perturbation step, the resetFlagOp operation involves both the strength units in setting the node strengths back to the charged state.

The destOp operation in step 5 above performs the relaxation operation on the source and destination nodes to give a new value for the destination node. During the logic update and perturbation steps only the flag unit is active during a destOp; the operation is performed only to schedule the destination nodes. During the blocking relaxation step, both the flag unit and the blocking strength units are active. The blocking strength is updated according to the source strength and the link strength. In the up/down relaxation step, three units are active: flag, up/down strength and logic state. The up/down strengths are set according to the source strengths and link strength with blocking from the destination blocking strength. Then the result of the strength computation is used to set the next logic state (NLS) field of the destination.

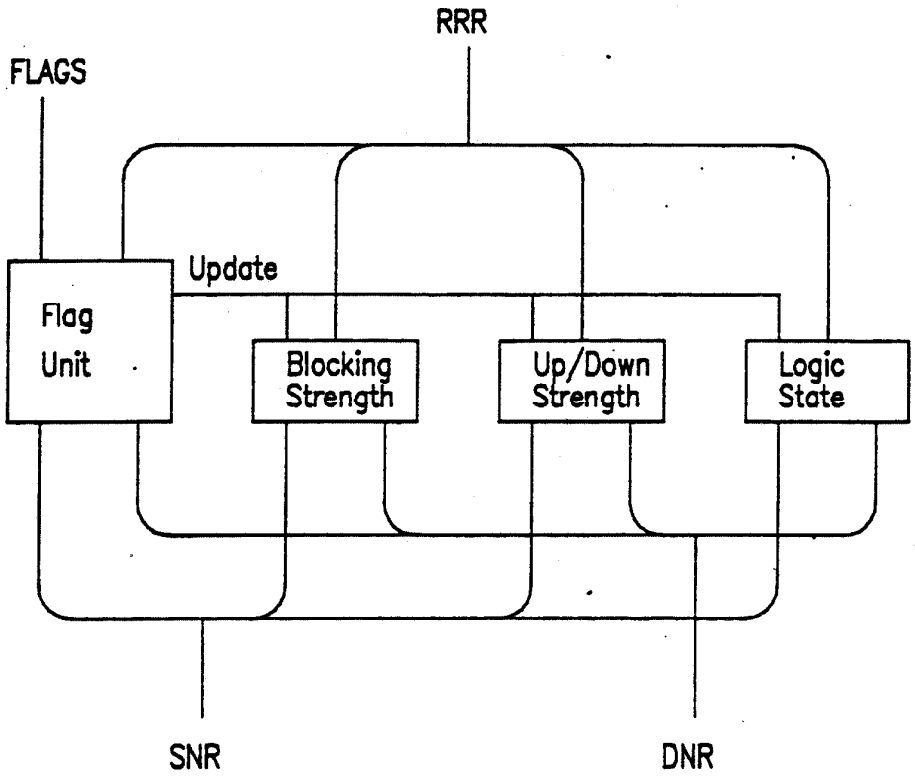


Figure 18. Relaxation Unit Block Diagram

The *flag unit* controls the updating of the R, U, L and P flags as well as producing two flags Update and LUpdate which are used in controlling the simulation process. During the resetFlagOp operation, the flag unit resets the activity bit corresponding to the current step (we will refer to this bit as decode(step)), and sets the update flag if the flag was altered.

Update ← Flags(step),
LUpdate ← Flags(nextStep),
Flags ← Flags * (not decode(step)) + update1*decode(nextStep),

The destOp operation varies with the simulation step. In general, the other field units control the setting of the Update and LUpdate lines. Update indicates a node has changed and should be written back and

rescheduled for this simulation step. LUpdate indicates that the node should be scheduled for the next simulation step. For the four simulation steps the values determining Update and LUpdate are shown in the following table:

STEP	UPDATE	LUPDATE
R	D.RSTR $\neq$ R.RSTR	D.RSTR $\neq$ R.RSTR
U	D.USTR $\neq$ R.USTR OR D.DSTR $\neq$ R.DSTR	R.LS $\neq$ R.NLS AND NOT L FLAG
L	NEVER	SIGNAL FROM NTU INDICATING L.TS CHANGED
P	NOT P FLAG	NOT R FLAG

Table 18. Update and LUpdate Flag Settings

The blocking strength relaxation unit selects one of four choices for the result RSTR field. During blocking strength relaxation when the destOp function is selected, r.RSTR $\leftarrow$ max(d.RSTR, min(s.RSTR, l.TSTR). A comparator sets the update flag if there were any changes between d.RSTR and r.RSTR. During all other destOps, r.RSTR $\leftarrow$ d.RSTR. During perturbation, when the resetFlagsOp is selected, r.RSTR $\leftarrow$ s.NSTR. During all other resetFlagOps, r.RSTR $\leftarrow$ s.RSTR. A block diagram of the RSTR field unit is shown in figure 19 below.

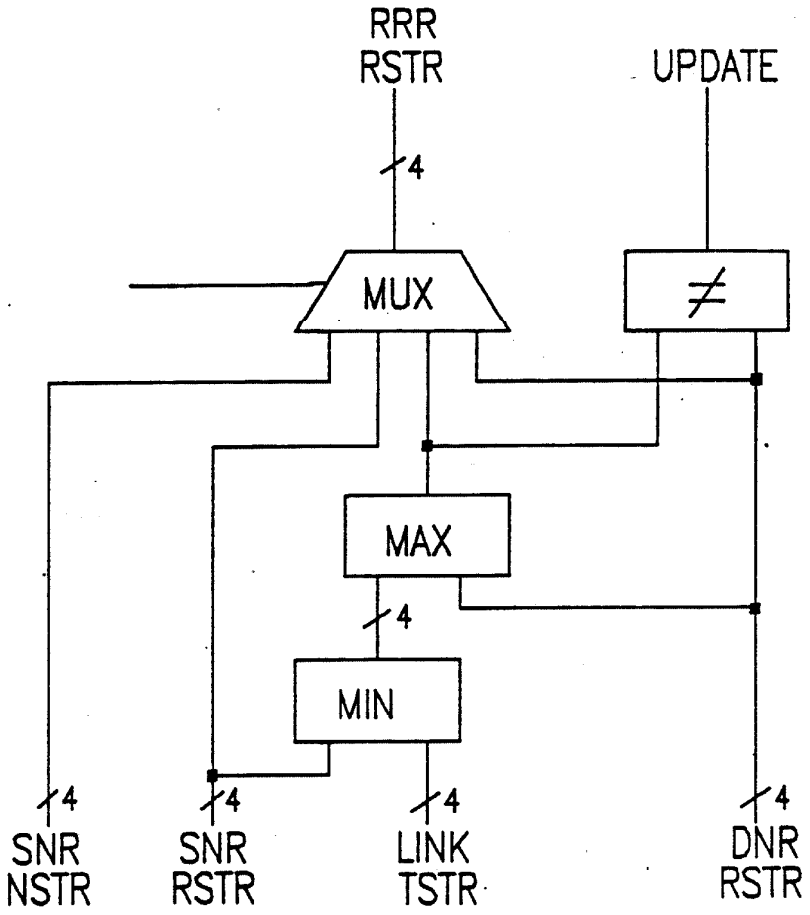


Figure 19. Blocking Strength Relaxation Field Unit

The Up and Down strength relaxation units are identical. The Up unit is shown in figure 20 below. It differs from the blocking strength unit only in the blocking function which prevents the USTR field from being updated to a value less than the blocking strength.

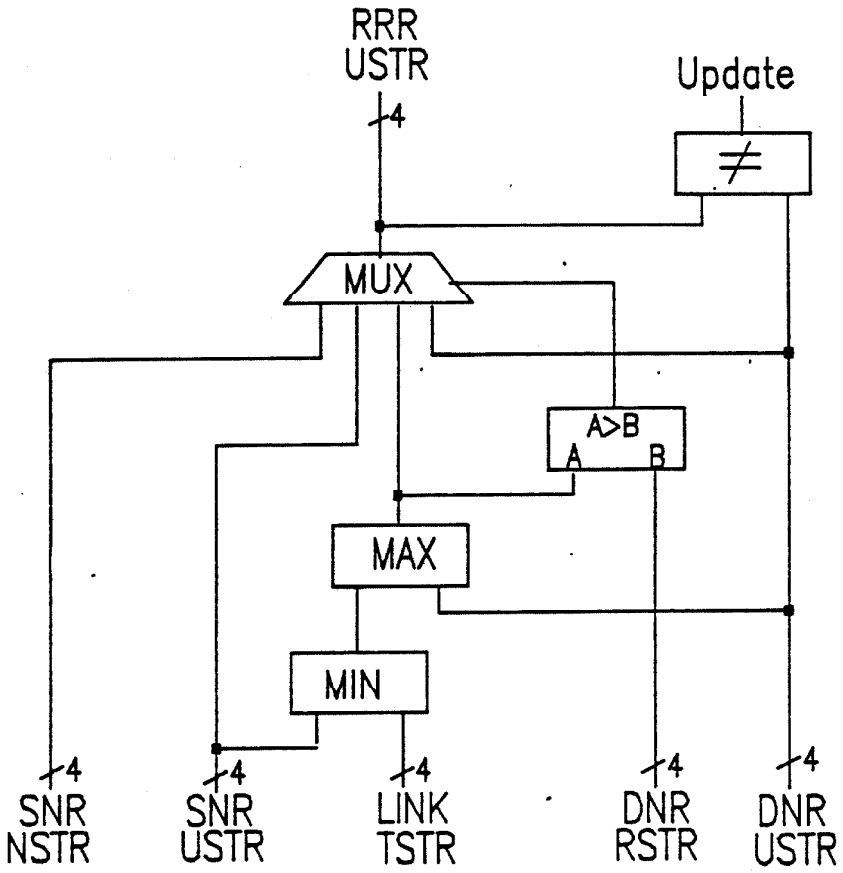


Figure 20. Up Strength Relaxation Field Unit

Figure 21 below shows the logic state field unit. The logic state field unit consists of a comparator which sets the new logic state depending on the three signal strengths. A multiplexor selects either this computed logic state or the source or destination for operations where the state does not change. The old logic state field can be set to the source, or destination old logic state fields or to the source new logic state field.

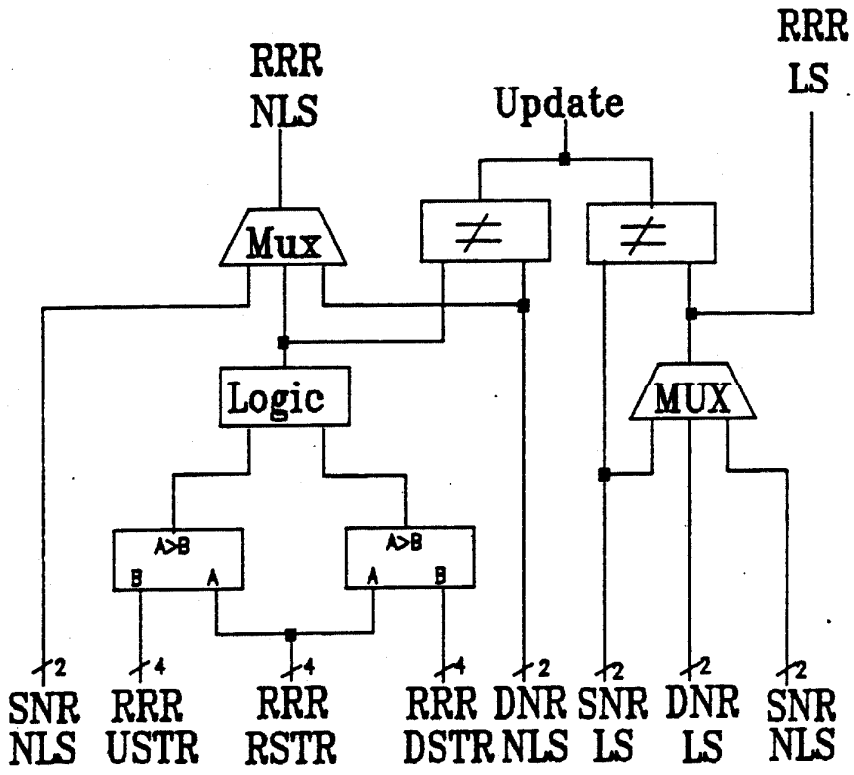


Figure 21. Logic State Field Unit

Network Traversal Unit (NTU):

The NTU provides destination node pointers and connecting link strengths to the NOU during relaxation steps and performs link state update during the logic update step. As shown in figure 22 below the NTU consists of a link and gate list memory (LGLM), incrementing gate and link list address registers (GAR), (LAR), an intermediate link and gate pointer (ILP), a link update unit (LUU), and a sequence controller. The NOU communicates with the sequence controller via a START input and READY and DONE outputs. The sequence controller also signals the IOU when an external link or gate requires a message to be transmitted to another SP. Since a single physical SP can be paged so that it contains several smaller virtual SPs, mapping registers are included to detect if an external link points to a virtual processor resident in the same SP. This comparison avoids sending unnecessary messages.

In operation the NOU loads the traversal counters, LAR and GAR, by reading the pointer fields of the source node from the NLM and signals the controller to START. The controller then begins the network traversal appropriate for the current simulation step.

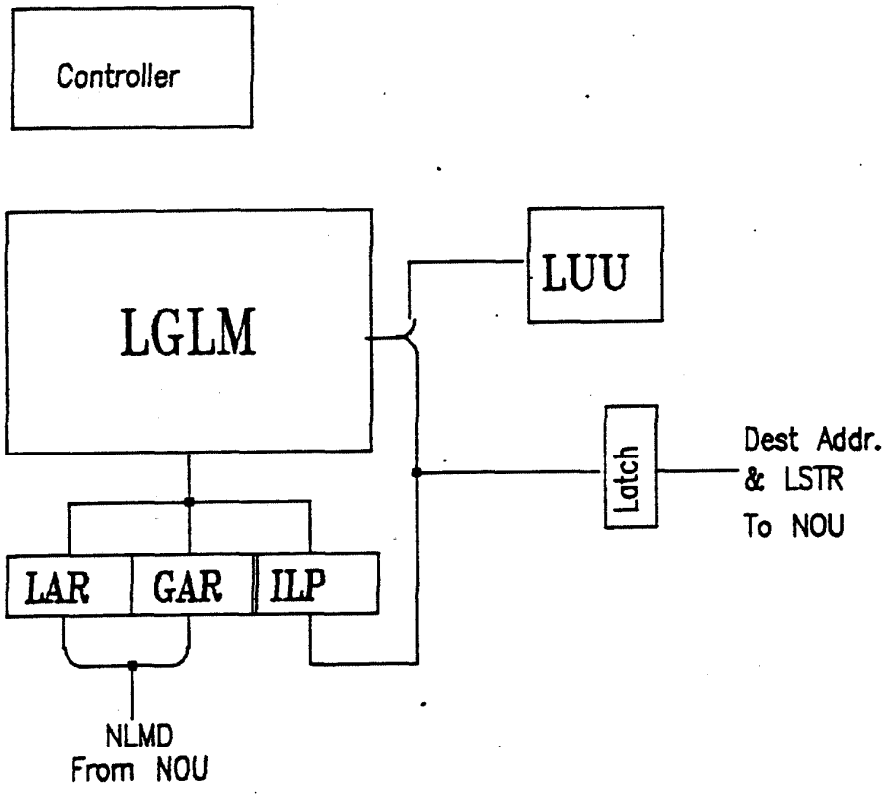


Figure 22. Network Traversal Unit

The NTU has two distinct modes of operation. During the R, U and P steps, the traversal unit operates in *relaxation mode*. *Update mode* operation occurs during the logic update step.

Relaxation mode operation involves sequencing through all of the links associated with the source node in the NOU's RU. Each *active* link is sent to the NOU for relaxation (the READY signal is asserted). The NTU looks one active link ahead of the NOU to minimize interlocking. When the last link of a node's list is encountered, the DONE signal is asserted. The sequence of events in relaxation mode is as follows:

1. Controller receives START signal,
   GAR and LAR are loaded.

2. LINK ← LGLM[LAR],
   LAR ← LAR + 1,
   if active(LINK) assert READY,
   if active(LINK) but no request then wait for request in state 3,
   if last(LINK) then terminate and assert DONE,
   if external(LINK) and no match in mapping registers then latch VPA and go to state 4.

3. if last(LINK) and active(LINK) and request then terminate and assert DONE,
   if active(LINK) and not request then repeat state 3,
   if active(LINK) and request and not last(LINK) then go to state 2.

4. LINK ← LGLM[LAR],
   transmit packet to IOU,
   go to state 2.

This sequence shows that relaxation mode operation involves simply sequencing and testing links. Communication with the NOU via the READY/ACK handshake interacts with the next(NTU) operation in the NOU procedure above. Nodes across which no relaxations occur are filtered out by the active(LINK) function which is true under the conditions shown in the following table:

STEP	CONDITION
R	LINK.TS = CLOSED
U	LINK.TS = CLOSED OR LINK.TS = X
P	LINK.TS = CLOSED OR LINK.TS = X

Table 19. Active Link Conditions

The last(LINK) function is true if the link being tested is of type LASTLINK.

Update Mode operation involves sequencing through the gate list of a node and updating the state of all links pointed at by gates. If the state of a link changes, the NOU is notified so the destination node of the link can be scheduled for perturbation. The sequence of events in update mode is as follows:

1. Controller receives START signal,
   GAR and LAR are loaded.

2. ILP ← LGLM[GAR].LP,
   GAR ← GAR + 1,
   if external(GATE) and no match in mapping registers then latch VPA and go to state 5,
   if last(GATE) then set LAST,
   if null(GATE) then terminate and assert DONE.

3. LINK ← LGLM[ILP],
   update LINK,
   LGLM[ILP] ← LINK,
   if LINK.TS changed but no request then wait for request in state 4,
   if LINK.TS changed and request then signal READY,
   if LINK.TS did not change and LAST then terminate and assert DONE,
   else go to 2.

4. if LAST, and request then terminate and assert DONE,
   if not request then repeat state 4,
   if not LAST and request then go to state 2.

5. GATE ← LGLM[GAR],
   transmit packet to IOU,
   go to state 2.

Scheduling Unit (SU): The scheduling unit maintains event stacks on which nodes are scheduled for each of the simulation steps. Sixteen stacks are maintained for the relaxation steps to implement stacks prioritized by node strength. Currently only single stacks are used for the logic update and perturbation steps; however the stack pointer memory (SPM) has room for 4096 stack pointers and hooks are provided in the architecture for implementing timing simulation by using separate logic update stacks for each time step.

The SU communicates with the NOU by means of push current, push next and pop current command

lines and ready, available and done status lines. The ready line indicates that the SU is ready to accept push commands. Asserting either or both of the push command lines when the SU is ready schedules the node address on the NLMA bus for the appropriate simulation step. The available line reflects the state of the SU pop ahead buffer. If available is TRUE, a node address is available in the pop buffer, if it is false, the NOU must wait before loading an address from the pop buffer into the SAR. When a pop address is available, the NOU signals the SU that the address has been accepted by asserting the pop command line. Note that push current, push next and pop may all be asserted in the same clock cycle. The done status line indicates when all stacks for the current simulation step are empty.

As shown in figure 23 below the scheduling unit consists of a stack memory (STKM) which holds all the scheduling stacks in a linked list data structure, a stack pointer memory (SPM) containing 4096 stack pointers addressed by stack pointer addressing logic (SPAL), a free register (FR) which points to the next unused location in the stack memory, data input and output registers (DIN), (DOUT), an empty comparator (EC), and a strength counter (SC).

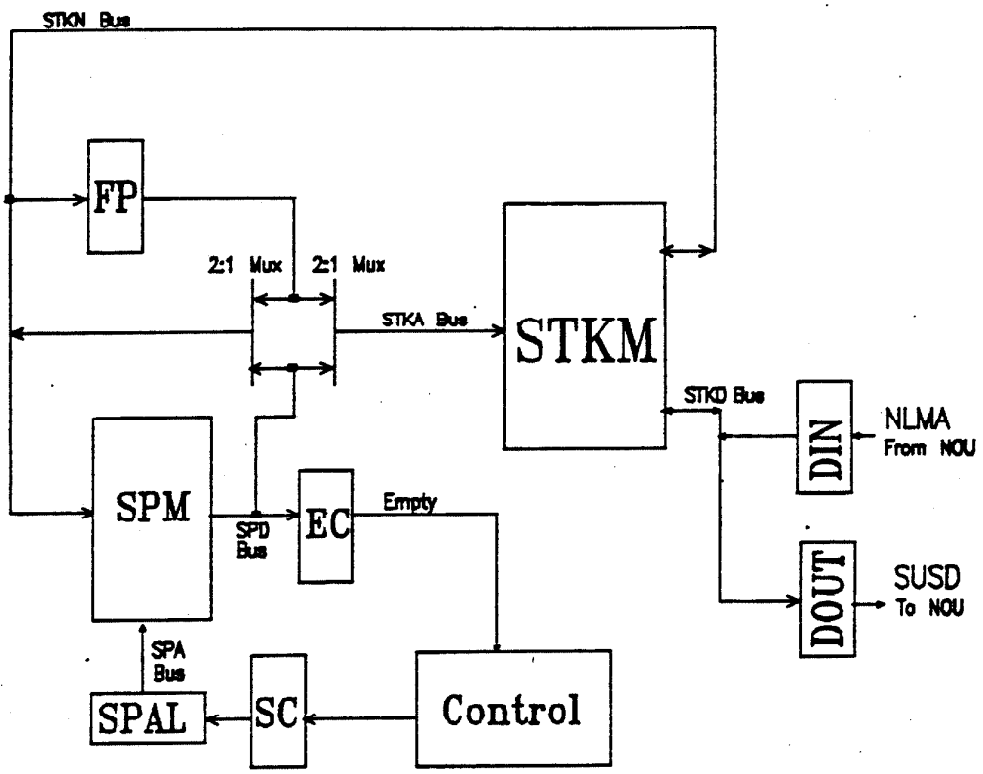


Figure 23. Scheduling Unit

During a push operation the SPAL calculates the address of the stack pointer based on the strength of the node and the current simulation step. The stack pointer is read from the SPM at the calculated address. At the same time the free register is used to address the STKM reading the first element in the free list. Then to insert the new data in DIN into the stack, the following data transfers are performed:

- FP ← STKM[FP].NEXT ;
- SPM[SPAL] ← FP ;
- STKM[FP].DATA ← DIN ;

▸ STKM[FP].NEXT ← SPM[SPAL] ;

A pop from a stack requires at least two cycles. First, the SPAL calculates the address of the highest priority stack pointer for the current simulation step. This stack pointer is read from the SPM and checked to see if it is NULL (stack empty). If a NULL stack pointer is found, the next highest priority stack is checked and so on until either a nonempty stack is found or the SU declares DONE. When a non-NULL SP is read from the SPM it is latched in a holding register and on the following cycle used as an address to the STKM, reading the top of stack data into the DOUT register. Then to remove this element from the top of stack the following transfers are made.

▸ FP ← SPM[SPAL] ;

▸ SPM[SPAL] ← STKM[SPM[SPAL]].NEXT ;

▸ DOUT ← STKM[SPM[SPAL]].DATA ;

▸ STKM[FP].NEXT ← FP ;

Because there are a large number of unused stack pointers, and the linked list data structure used to implement the stack is very flexible, other scheduling algorithms can be implemented on the machine by changing the SU controller. Consider the timing simulation event-list data structure shown in figure 24 below. When an event is to be scheduled, a node record off the free node list is allocated and a search of the time list is begun to match the schedule time. If an existing time record is found, the new node record is appended to the node list pointed at by that time record. Otherwise, a new time record is allocated, inserted in the list and the new node record becomes the time record's node list. If a node must be unscheduled, a node record is deleted from a node list and placed on the free list. Retrieves are performed from the current time node list.

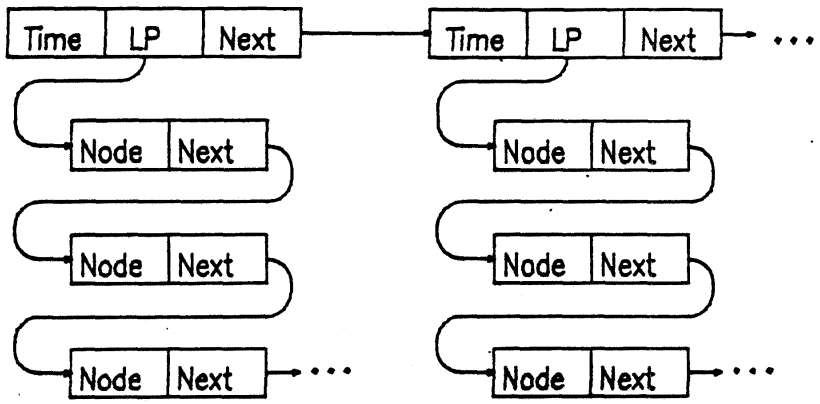


Figure 24. Event List Data Structure

Balance and Utilization in the SP: SP operations are divided into two phases: the four cycle initialization for each new source node, and the steady state operation performing relaxations against each destination node. In both phases, the NOU is utilized 100%of the time unless it blocks for a node pointer from the NTU. Typically blocking will yield a NOU utilization of 75%. In general, the NOU will not block on pops from the SU because the SU has ample time to pop ahead during the first two cycles after a new source node is read from the pop ahead buffer. In the relaxation phase, the NOU operates at a rate of 1 relaxation per cycle after a four cycle initialization phase. During the initialization phase, the scheduling unit performs one retrieve operation giving a utilization of 50%. In the relaxation phase, one or two schedule operations are performed per relaxation giving a utilization of between 50%and 100%. The NTU is utilized close to 100%of the time depending on the percentage of active and inactive links. If an inactive link is encountered, the NOU will idle waiting for an active link from the NTU.

With all three functional units utilized more than 50%of the time, the SP is a well balanced multiprocessor. This balance was achieved by adding specialized circuitry, the RU, to the NOU. Without the RU, the NTU and SU would be utilized about 3%of the time as the NOU would take 30 cycles to perform a single relaxation.

5.2 Message Switch (MS)

The message switch, shown in figure 25 below, stores and forwards messages between MSE processors using a message queue. The switch consists of a microcoded engine (MCE), a queuing memory (QM) which holds messages for swapped out processors, a mapping unit (MAP) which translates virtual processor addresses to physical addresses or queue addresses, an input FIFO which buffers messages from the message bus to the switch, and an output port to the message bus. Messages to be enqueued are transmitted to the message switch over the message input bus. A daisy chained bus access mechanism arbitrates multiple simultaneous requests for the bus. Messages transmitted over the bus (48 bits at a time) are enqueued in the MS's input FIFO. If the destination of a message is a virtual processor which is swapped in, the processor MAP forwards the message directly to the output bus with a translated physical processor address. The microcode engine handles messages for swapped out processors by placing them in the appropriate message queue.

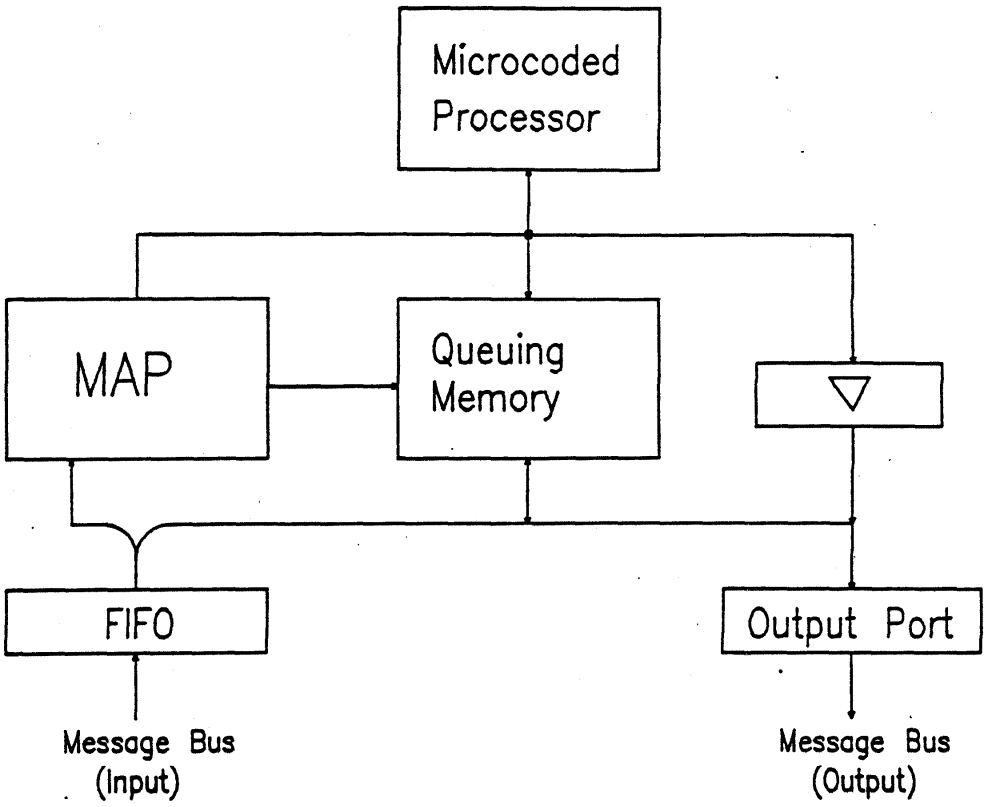


Figure 25. Message Switch Block Diagram

When the input FIFO is empty, the microcode engine attempts to empty the message queues of those processes which are currently assigned to physical processors. Messages are dequeued from the appropriate message queues and written over the message output bus to the destination processor until a message is refused because of a full input FIFO.

Traffic in the message switch is dependent on the size of the SPs and the number of SPs connected to the switch. The message transmission time is one clock cycle. If we assume that the percentage of relaxations which require message transmission is equal to one minus the static locality of the circuit, then the message switch will become saturated when connected to a number of processors equal to twice the reciprocal of this percentage. For instance, with 1024 nodes per processor, the static gate locality is 78.5%and eight processors may be connected before the switch will be saturated during the logic update step; however, only 10%of the simulation time is spend in logic update. For the other three steps, the static connection locality determines the saturation level. For 1024 nodes per processor, the connection

locality of the MOSAIC chip from section 4.3 is 100%so there would be negligible message traffic.

5.3 Auxiliary Processor (AP)

One or more auxiliary processors (APs) may be added to a MSE to improve the performance of functional simulation. The AP as shown in figure 26 consists of a microcoded engine and a 68000 microprocessor which share the functional simulation tasks.

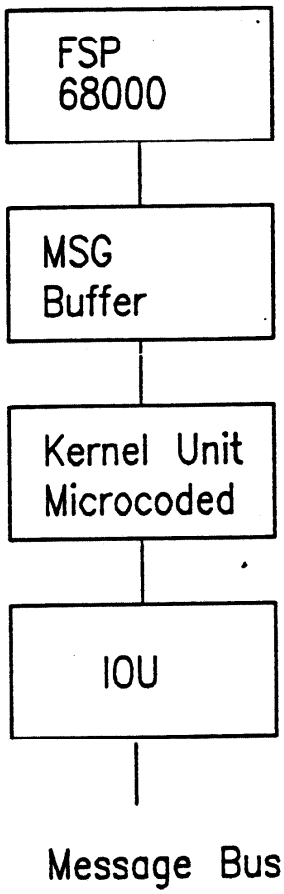


Figure 26. Auxiliary Processor Block Diagram

The microcode engine is responsible for:

1. receiving logic update messages from the message switch and updating input variables of each functional block;

2. scanning the sensitivity list for each input variable change to trigger the evaluation of a functional block when a sensitized input changes;

3. transmitting changed output variables to message switch after evaluation of functional block;

4. performs functional simulation of simple blocks such as RAMs, ROMs, PLAs and logic gates.

The 68000 microprocessor is used to perform the functional simulation of complex blocks. It communicates with the microcoded engine through a message buffer in shared RAM. After a logic update cycle, the microcoded engine writes the functional blocks to be evaluated into the message buffer. When the evaluation of a functional block is complete, the microprocessor signals the microcode engine so it can queue up the changed signals for transmission during the next logic update cycle.

5.4 Host Processor (HP)

The host processor is a SUN workstation which acts as the user interface to the MSE and the global MSE controller. The functions of the host processor are:

1. providing a user interface: performing command interpretation and displaying simulation results ;

2. sending logic update messages for each input vector, and receiving messages to compose output vectors.

3. loading and partitioning the circuit model ;

4. coordinating the operation of the SPs, MS, and APs ;

5. performing debugging and system diagnostics.

Chapter 6

Design

A Mossim Simulation Engine subnetwork processor is constructed from 377 integrated circuits, 241 (LS,S,AS)-TTL logic parts, 133 fast MOS static memory parts and 3 bipolar ROMs packaged on a single wire wrap board. The design is described as a DSIM connectivity file (1500 lines), and PAL programs for the 34 PALs in the design. This chapter presents some of the design details to give the reader an idea of the design style used. A complete description of the design is beyond the scope of this report.

The next section presents an overview of the design. The timing methodology is presented in section 6.2. Section 6.3 describes the three-level control structure used in the MSE. A list of the parts used in the design is presented in section 6.4. Beginning with section 6.5 some example schematics are reviewed. The SU is presented in section 6.5, the NTU in section 6.6, the RU in section 6.7, and the IOU in section 6.8. The host bus interface, addressing logic and global status circuits are described in section 6.9. This chapter concludes with a discussion of the lessons learned in the detailed design of the MSE and the things which would be done differently if the design were to be done again.

6.1 Design Overview

In this section the high-level design decisions made early on in the design of the MSE are described: technology selection, subnetwork processor size, and resource sharing.

Technology Selections

The original MSE concept was to develop an architecture where each SP was implemented as a single MOS integrated circuit. As the design progressed, however, it became clear that the SP was far to complex to be implemented on a single chip. Given the complexity of the SP, two design alternatives seem feasible: implementation in TTL-PAL logic, or implementation as a custom VLSI chipset.

Using TTL-PAL technology the function units are implemented with TTL MSI logic circuits and PALs for those functions not available in MSI. The memories are implemented using fast (45ns) NMOS static RAM chips. This approach has the advantage of flexibility at the expense of high chip count.

During the prototype stage, however, this high chip count is worth the flexibility in making logic changes that this implementation affords. Design changes can be made by programming a new PAL, or moving a few wire wrap wires. This technique allows corrections to be made in a matter of minutes rather than the months required to iterate a VLSI design. For these reasons TTL-PAL implementation technology was chosen for the MSE.

If the design were stable, a VLSI implementation would be superior on account of its relatively low chip count. Each of the function units (excluding memory) could be implemented in at most two 40-pin packages, and if larger pinout packages (eg. 84 or 125 pin grids) were available, each unit could be implemented on a single chip. All flexibility need not be sacrificed if the chips are externally microcoded, or at least some of the microcode is external.

Subnetwork Processor Size:

The number of nodes which can be simulated on one SP determines the amount of subnetwork concurrency possible as well as the traffic through the message switch. Factors influencing the choice of SP size include:

- the balance of simulation speed to input and output vector speed ;
- concurrency issues: the traffic capacity of the message switch, and the swapping time ;
- the available sizes of commercial memory parts ;
- the balance of cost between logic and memory for a single SP.
- the number of chips which will fit on one board.

As long as external memory parts are used, item 3 above sets a minimum size of the SP 1024 nodes (using 2Kx8 memory parts for the NLM). Using this *minimum* number of nodes, 8K gates and links would be required (using 4Kx4 memory parts) to meet the 6:1 gate or link to node ratio, and a 4K stack memory is required. The total number of memory chips for a minimum size configuration (excluding the local RAM) is 20. However, it would not make sense to use 20 memory chips and 241 logic chips. As stated above, a balance should be achieved between the cost of memory and the cost of logic. Since the incremental cost of doubling the memory size is low compared to the total cost, it makes sense to

increase the memory size. A size of 4096 nodes would achieve approximate balance (136 memory chips vs 241 logic chips), and gives a total of 377 chips which fits on one large wire-wrap board. For these reasons a size of 4K Nodes, 32K gates and links was chosen for the MSE.

Since the first two items above may require a smaller processor size, the MSE SP has been designed to that a 4096 node SP may be sub-divided to hold 4 1024-node virtual processors simultaneously. A mapping register in the NTU checks external links to see if they are resident in the same SP and does not transmit a message for these local-external links.

With the VLSI implementation, the minimum memory size would achieve a fair logic-memory balance, however items 1 and 2 above may require a larger processor.

The maximum simulation speed which can be used is determined by the input and output vector rates as indicated in item 1 above. The simulated performance of a single SP MSE is 250K logic update events per second (or 200 to 500 times the performance of the MOSSIM simulator on a DEC VAX 11/780). If we assume that only 10%of all circuitry in the simulated circuit is active, and the maximum sustainable input/output vector rate is 1K vectors per second, then the maximum usable performance is $1K x 0.1N$ where N is the number of transistors in the circuit. Thus a circuit of 10^5 transistors would imply a maximum usable performance of 10M logic updates per second. This performance could be achieved by 20 to 50 SPs with each SP simulating from 2K to 5K transistors. Thus, the figure arrived at above of 4096 transistors per SP appears to be in the usable range.

It should be noted that the maximum input/output rate of 1K vectors per second assumed in the paragraph above does not apply in cases where functional simulation is used to generate the input/output vectors. In these cases, vector rates up to 100K vectors per second are possible although difficult to achieve. This subject is discussed in more detail in chapter 7.

Item 2 above is concerned with the traffic through the message switch. While this traffic is in part determined by the size of an SP, it is more directly influenced by the size of the message bus and the number of SPs on a message bus. The swapping traffic is determined by the swapping speed and the number of relaxations performed between swaps. These design parameters are discussed in the following paragraphs.

Message Bus Considerations

Traffic on the message bus is determined by two factors: 1) the number of SPs, n_s, and 2) the fraction of the complete circuit contained in a single SP, f_c. As noted in section 4.3, the gate locality of a circuit is considerably lower than its connection locality. Thus, the message traffic during the logic update step

will be higher than during the relaxation or perturbation steps. If we consider the percentage of logic updates which require messages proportional to the gate locality, then from the empirical expression for gate locality in section 4.3 we can calculate an expression for message traffic, t_p, (as a fraction of capacity) contributed by a single SP as :

$$t_p = 0.5\left(1 - f_c^{1/8}\right)$$

The constant, 0.5, in this equation reflects the fact that a message only takes one cycle to transmit, while a logic update requires two cycles. If we assume that all processors are busy, then the total message traffic, t_m, is given by, $t_m = n_e t_p$.

Three design variables are used to balance the message bus capacity vs the offered traffic: 1) size of an SP, 2) number of SPs on the message bus, 3) speed of the message bus vs an SP. If we fix the size of an SP at 4096 nodes and the speed of the message bus at twice the logic update rate of an SP, then the maximum number of SPs on the message bus can be calculated as a function of circuit size, N. If we note that $f_c = \frac{1}{\lceil \frac{N}{4096} \rceil}$ and set the traffic to one we get:

$$n_e = \frac{2\lceil \frac{N}{4096} \rceil^{1/8}}{\lceil \frac{N}{4096} \rceil^{1/8} - 1}$$

Note that a number of processors greater than $\lceil \frac{N}{4096} \rceil$ makes not sense. With these two limits on number of processors in mind, the following table gives tha maximum number of SPs on a single message bus before bus saturation for various circuit sizes:

CIRCUIT SIZE	NUMBER OF SPS
4K	1.00
8K	2.00
16K	4.00
32K	8.00
64K	6.83
128K	5.69
256K	4.93
512K	4.39
1M	4.00
2M	3.69
4M	3.44
∞	2

Table 20. Maximum Number of SPs vs. Circuit Size

The data in this table indicate that more than eight SPs on a message bus are almost guaranteed to saturate the message switch, while four processors on the bus provide a wide useful range (from 16K to

1M nodes) without saturation. This data suggests a hierarchical message bus structure for large MSE configurations with a fanout of four to eight at each level of the hierarchy as shown below:

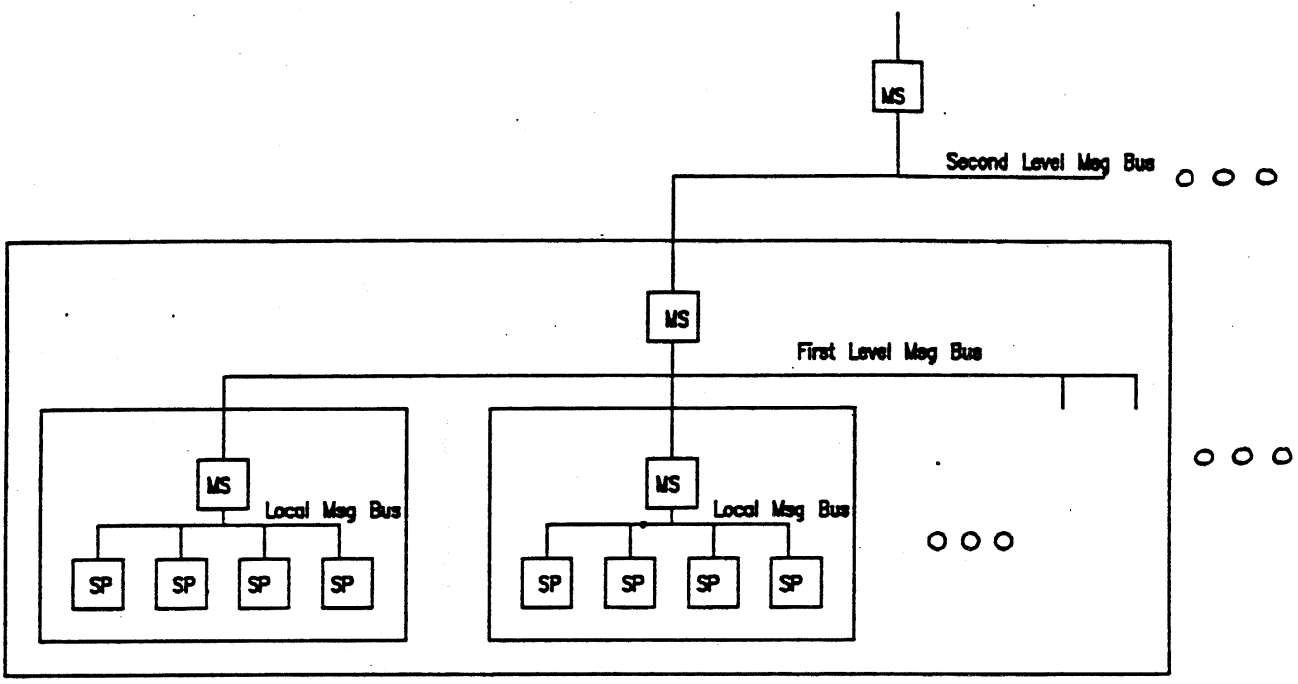


Figure 27. Hierarchical Message Bus

Before concluding this section, a few observations are in order:

1. The expression derived above for message traffic and number of SPs assume that traffic is proportional to $1 - l_g$ this assumption has not been tested. It is expected that message traffic will actually be much lower than this figure since active locality tends to be much higher than static locality.

2. If the number of SPs on a message bus is selected to prevent the message switch from saturating during logic update, the switch will be under utilized during the other simulation steps. Since only 10%of the simulator's time is spent in logic update, a more effective approach might be to select a number of SPs which saturates the switch during logic update, but makes better use of the switch during the relaxation and perturbation cycles.

3. Having a large number of SPs which saturate the message bus is better than having too few SPs to

hold a working set for the circuit. Having too few SPs will result in thrashing of virtual processor swaps and very low performance.

4. The tree interconnect presented above is not necessarily the best interconnection topology for message passing in an MSE.

Swapping:

As the prototype MSE SP has no Local RAM, swapping is expected to be quite slow. A 4K node, 16K transistor circuit contains 136K 16-bit words of state and connectivity information. Transferring this circuit over the host bus at $1\mu s$ per word requires 136ms. A complete swap (one VP out and another VP in) would require over a quarter of a second. Clearly a 4K node circuit is too large for swapping (even with a local RAM). A 1K node circuit requires 1/4 the time or about 60ms to swap. This swapping time is more reasonable. If a sufficient number of SPs is used to hold a working set of subnetworks swapping should not dominate the machine's performance. Adding mapping registers to the NTU allows the size of a virtual subnetwork processor which affects swapping performance to be different than the size of a physical subnetwork processor which affects message traffic.

Hardware Support for Timing and Circuit Simulation:

All the special purpose hardware described in chapter 5 is geared toward switch-level unit-delay simulation. The intention is to optimize the performance of the switch-level simulator while retaining the capability for timing and circuit simulation. While the functions of the SU and the NTU do not change significantly for timing and circuit simulation, the NOU must perform a considerable number of arithmetic operations making it a bottleneck in the simulation process. The only means for performing these operations with the proposed design is to execute them in the CP's data path. Adding special purpose arithmetic hardware to the NOU would considerably accelerate circuit and timing simulations, however, for the prototype design such hardware will be omitted.

Resource Sharing

If two components of the SP are rarely used at the same time, they may be combined without degrading performance. An example of resource sharing is the combining of gate list and link list memories. The gate list memory is used only during the logic update step. By combining the gate list memory with the link list memory, the overhead of separate memories, data and address busses is eliminated. The penalty is that an extra cycle is added to each logic update. With the combined memories, two cycles are required per update rather than one with separate memories. However, this slowing of the logic update step helps balance traffic through the message switch which is greatest during this step.

Other potential opportunities for resource sharing exist in the relaxation unit where only a few of the field units is active at a time. However, the cost of multiplexing required to share the units exceeds the cost of the units themselves.

6.2 Timing Methodology

The Mossim Simulation Engine operates off of a 20MHz master clock which drives a programmable clock generator. Under control of the host processor, the frequency and duty cycle of the system clock can be varied over a wide range. The nominal system clock frequency is 5MHz giving a major cycle time of 200ns. Each major cycle is divided into two minor cycles or phases ϕ_1 and ϕ_2.

During ϕ_1 (clock high), data is read from memory and computations on this data are performed. During ϕ_2 (clock low) the resulting data is written back to memory.

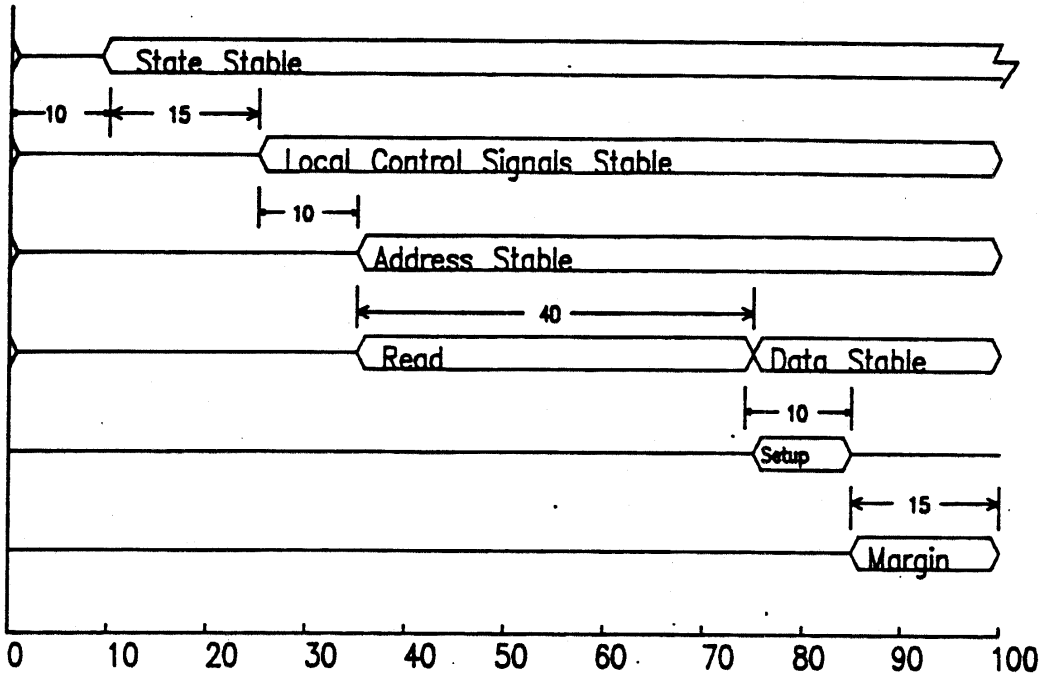


Figure 28. MSE Timing

The function units of the MSE are designed around this two phase methodology. A typical timing budget is shown in figure 28 above. During ϕ_1, following the rising edge of the clock, time is allocated as follows:

- *10ns (0-10ns)*: State transition, the control processor's microword and the local PAL controllers state variables are changing. (74AS374 clock to data and AMD A-series registered PALs).
- *15ns (10-25ns)*: Local decode, the control decoders generate control signals from the new state information. Since writes are not allowed during ϕ_1, no information is lost by control glitches between the rising edge of the clock and the state becoming stable. (AMD A-series PALs).
- *10ns (25-35ns)*: Address select, an address is enabled onto each memory's address pins.
- *40ns (35-75ns)*: Memory read, data at the selected address is read from each memory. (InMOS 1400-45, 1420-45).

- *10ns (75-85ns)*: Set-up time, data read from memory is latched into holding registers (74AS374).
- *15ns (85-100ns)*: Margin.

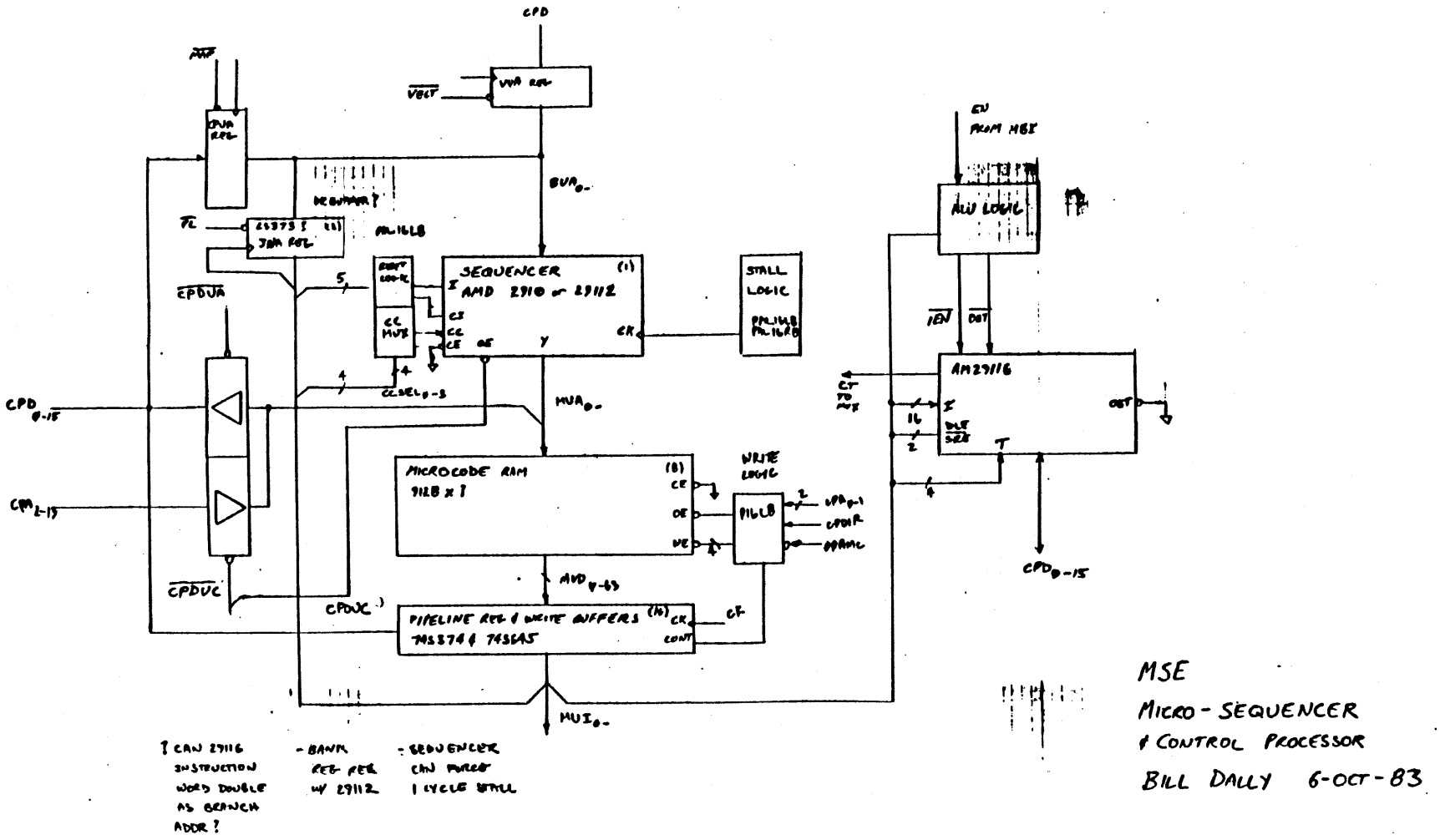
Similarly, during ϕ_2, following the falling edge of the clock, data is written back to memory.

- *15ns (0-15ns)*: Local decode, note that no state transition is required as state variables do not change on the falling edge of the clock. However, new control signals are generated.
- *10ns (15-25ns)*: Address select.
- *45ns (25-70ns)*: Memory read.
- *10ns (70-80ns)*: Set-up Time.
- *20ns (80-100ns)*: Margin.

One part of the machine which does not follow this timing discipline is the relaxation unit. The RU gets its data 10ns after the rising edge of the clock and has until the next rising edge to produce a result (200ns). The actual worst-case delay through the RU is 150ns.

6.3 Control Architecture

There are three levels to the MSE control architecture. At the top level, a microcoded controller signals the NOU to perform relaxation operations using data either from the scheduling unit or the input FIFO. The microcode engine also includes an AMD29116 data path for performing housekeeping operations and has access to all the memories in the MSE. In the course of performing these operations the NOU may request the services of the SU or NTU by signaling their local controllers. A schematic of the microcoded engine is shown in figure 29 below.



? CAN 29116 INSTRUCTION WORD DOUBLE AS BRANCH ADDR ?

- BANK REG REG W 29112

- SEQUENCER CAN FORCE 1 CYCLE STALL

Figure 29. Schematic: Microcode Engine

The local state machine controllers in the SU and NTU form the second level of control. They allow the SU and NTU to operate autonomously from the sequence followed by the microprogram. In effect each unit can proceed at full speed until it blocks for data from another unit. To interlock these interacting function units a stall mechanism is implemented which stalls a unit when it is awaiting service.

The bottom level of control is local PAL decoders in each of the function units. These PALs decode the local state (from the PAL state machines and microcode) and generate strobes and control signals to read/write memory, steer data, and select the function of each data path. For speed, all local decode is implemented in one level of logic using fast AMD A-series PALs (15ns delay).

A typical MSE control sequence is shown below in figure 30.

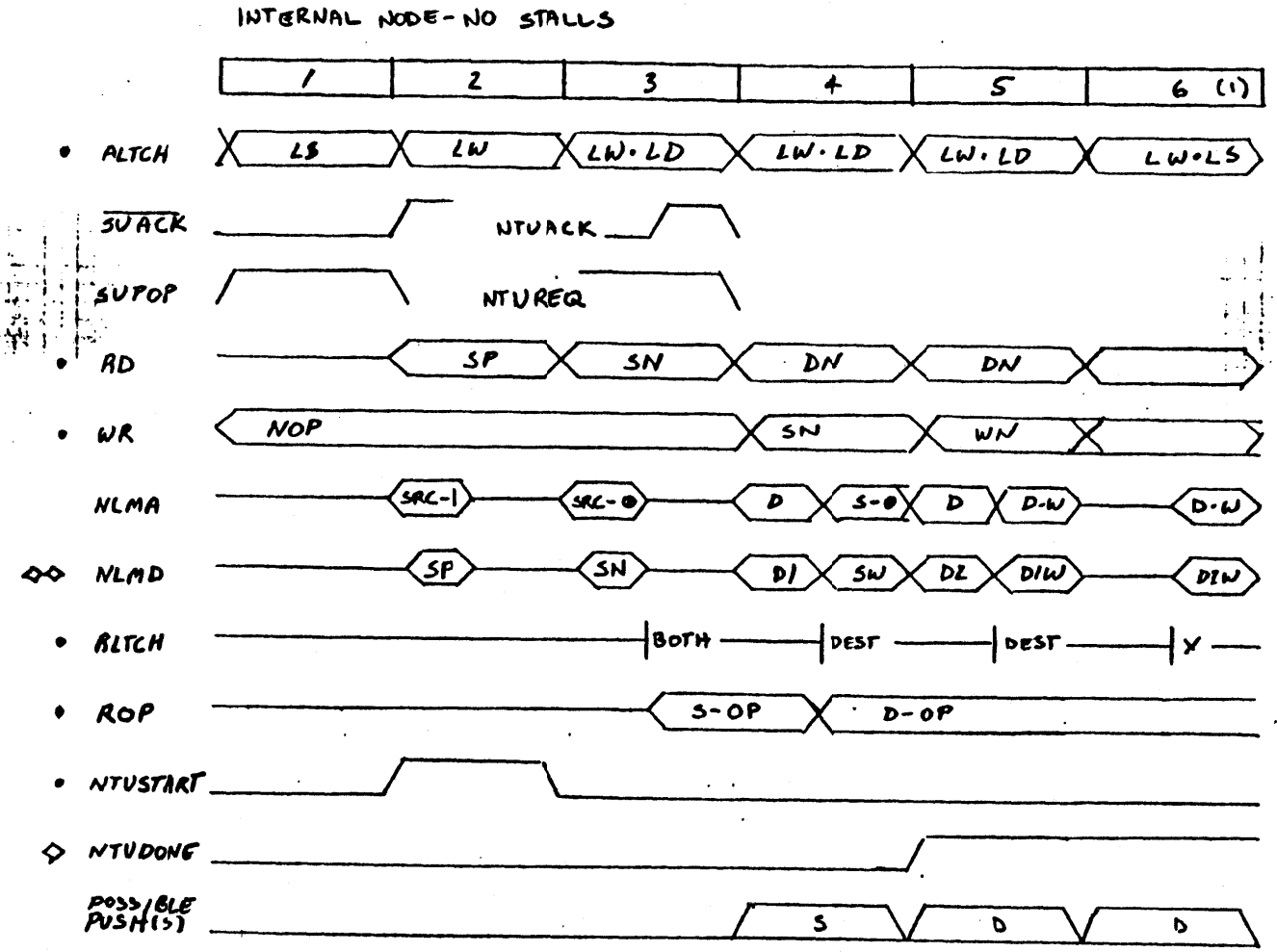


Figure 30. Typical Control Sequence

6.4 Bill of Materials

The following integrated circuit chips are used in the MSE:

PART NO.	DESCRIPTION	QUANTITY
PAL16R8A	8-INPUT 8-OUTPUT REGISTERED PAL	7
PAL16L8A	10-INPUT 8-OUTPUT COMBINATIONAL PAL	28
74AS374	OCTAL D FLIP-FLOP	75
74S373	OCTAL D LATCH	21
74S244	OCTAL BUS BUFFER	32
74S645	OCTAL BIDIRECTIONAL BUS BUFFER	18
74LS688	OCTAL COMPARATOR	1
DIPSW	8 POSITION DIP SWITCH	1
74S06	HEX OPEN COLLECTOR INVERTER	1
74AS04	HEX INVERTER	2
74LS461	EIGHT BIT COUNTER	8
74S161	FOUR BIT COUNTER	1
74S85	FOUR BIT COMPARATOR	8
74S133	THIRTEEN INPUT NAND GATE	5
74S139	DUAL 2 → 4 DECODER	1
74LS453	QUAD 4 → 1 MULTIPLEXER	5
74S151	8 → 1 MULTIPLEXER	2
74S157	QUAD 2 → 1 MULTIPLEXER	3
AMD2910	MICROSEQUENCER	1
AMD29116	DATA PATH	1
C67401	64-WORD BY 4-BIT FIFO	20
TOTAL		241

Table 21. Logic Chips used in the MSE

PART NO.	DESCRIPTION	QUANTITY
IMS1420-45	4K-WORD BY 4-BIT 45NS RAM	28
IMS1400-45	16K-WORD BY 1-BIT 45NS RAM	97
4108-55	1K-WORD BY 8-BIT 55NS RAM	8
27S43-45	4K-WORD BY 8-BIT ROM	3
TOTAL		136

Table 22. Memory Chips used in the MSE

The integrated circuits are packaged on a 15 inch by 15 inch wire-wrap board (MuPack 347 series, part number 347886-01) composed of 68 rows of 136 wrap posts for 9248 total posts. The wire list and wire-wrap command file for the board were compiled using the DSIM physical design subsystem [24]. Up

to 13 boards are packaged in a card cage (MuPack 3314450-03). The card cage, a 1000W power supply, Sun workstation (less monitor), and cooling fans are housed in a 19 inch cabinet.

6.5 Scheduling Unit

The schematic diagram of the scheduling unit is shown in figure 31 below. The scheduling unit control and stack pointer addressing logic are on separate schematics. Both the stack pointer RAM and the stack RAM are implemented with InMOS 1420 4K $\times$ 4 static RAM chips. The free register (FR) is implemented using 74AS374 octal D flip-flops. The FR is duplicated so it can be enabled onto either the STKA bus or the STKN bus with no more than the 10ns delay allocated for enabling data and addresses onto busses. Also shown in the schematic are the bus buffers which enable the control processor to gain access to the memories and the free register.

The scheduling unit is representative of all the MSE function units in that is contains a great deal of data steering logic in the form of octal bus interface parts: 74AS374, 74S373, 74S244 and 74S245. If the function units of the MSE were integrated onto custom ICs these interface parts, which now account for more than half the logic chips on the MSE SP board would, occupy a negligible amount of area.

The SU also has many local control lines. The SU controller has three PAL16L8A decoder pals driving 20 control lines. If the SP were integrated onto a custom chip, the controller could be combined into one or two medium sized PLAs.

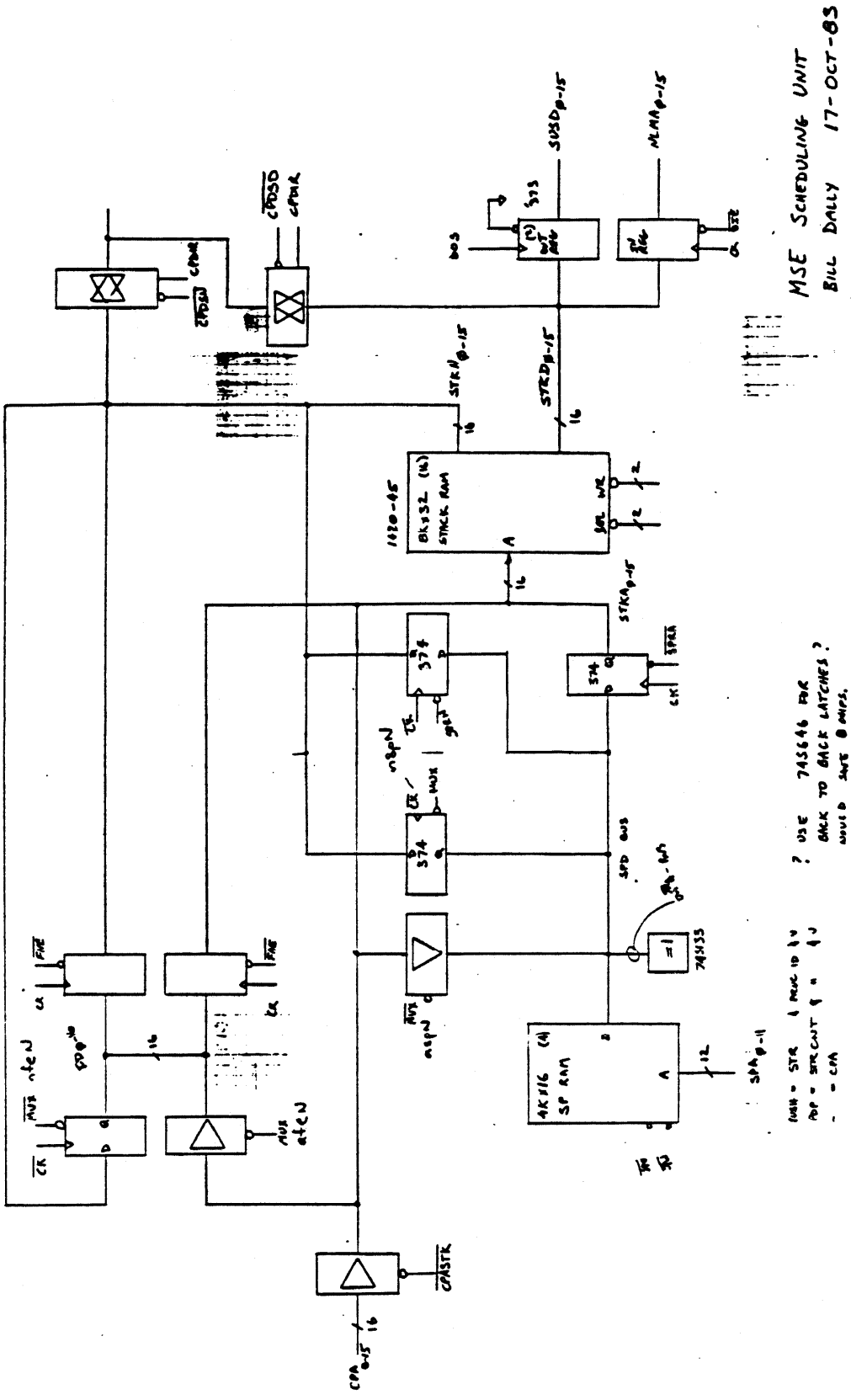


Figure 31. Schematic: Scheduling Unit

6.6 Network Traversal Unit

The Network Traversal Unit is shown in figure 32 below. The controller and mapping logic are on separate schematics. The NTU is memory intensive composed of 96 InMOS 1400 chips in the LGLM. Because the 1400 has separate data input and output lines 74AS374 registers are used to latch the data read during ϕ_1. In a logic update cycle, this same data is written back during ϕ_2 along with the new transistor state.

This memory is addressed by the four registers at the bottom of the schematic. The link and gate counters (LAR,GAR) are implemented with two Monolithic Memories 74LS461 octal counters each (parallel load with tri-state outputs). The ILP latch is implemented with 74AS374 registers, and the control processor address can be gated on the address bus via two 74S244 bus buffers.

As with the SU, the non-memory portion of the NTU could easily be integrated on one high-pinout custom IC.

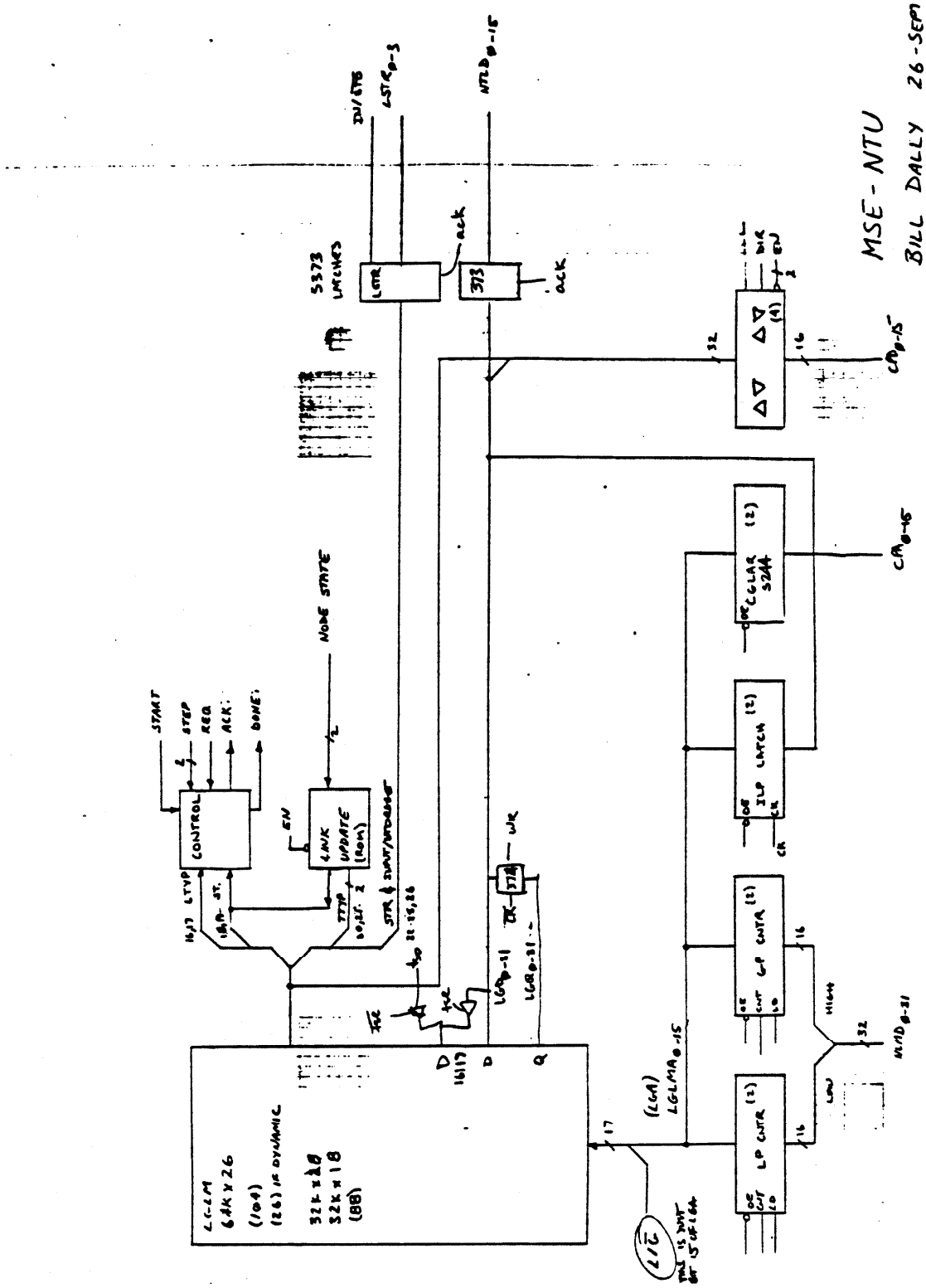


Figure 32. Schematic: Network Traversal Unit

6.7 Relaxation Unit

The relaxation unit shown in figure 33 below is how the MSE attains a speedup of 200-500 compared to conventional computers. The RU computes in 100ns typical 150ns maximum what requires upwards of 30 lines of Mainsail code[34].

The logic of the RU is implemented to a large extent using programmable logic (ROMs and PALs). Three identically programmed ROMs are used to implement the relaxation max-min function, and PALs control logic update and switch the multiplexers based on update results and simulation step. Comparators are used to perform the blocking operation.

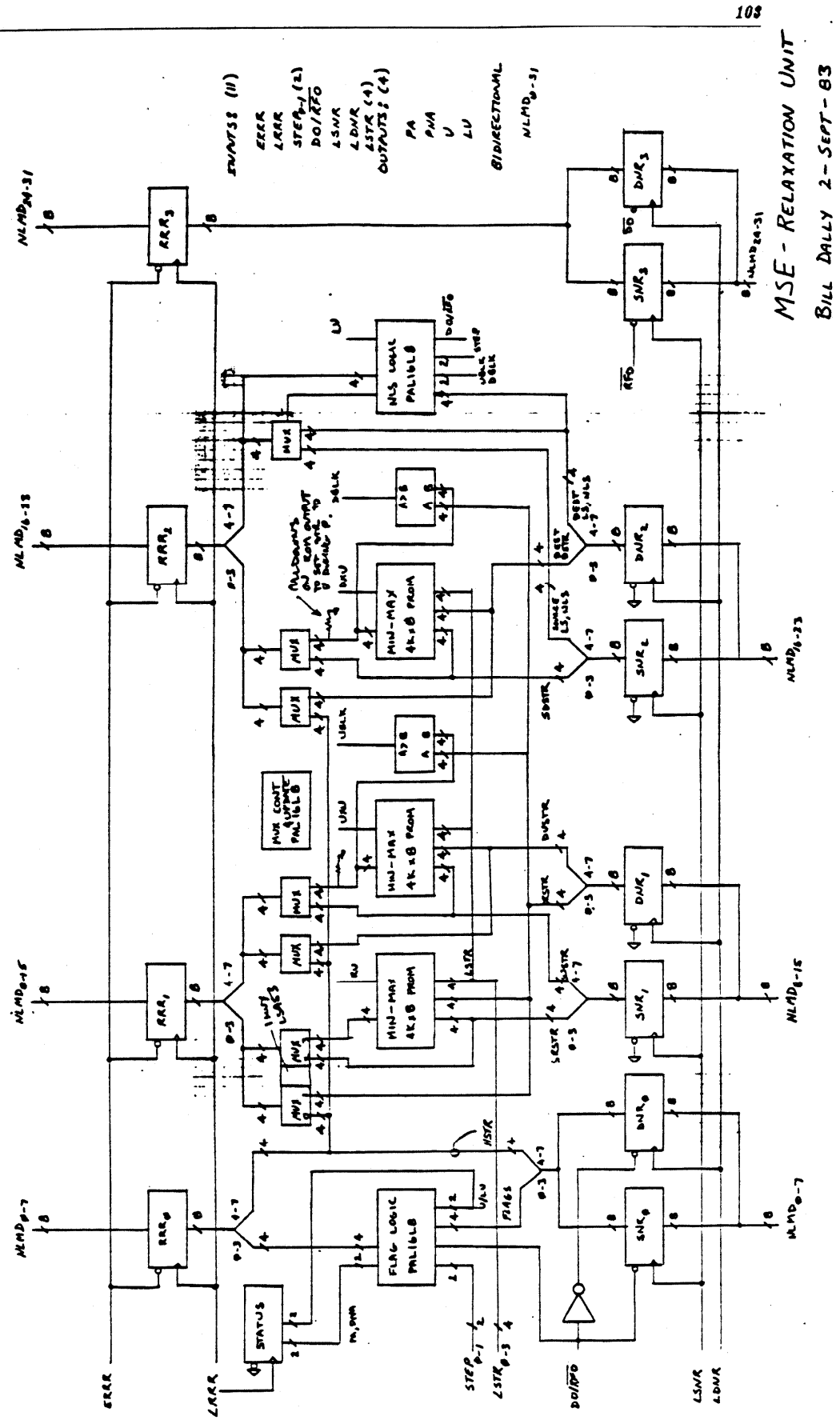


Figure 33. Schematic: Relaxation Unit

6.8 Input/Output Unit

The Input/Output unit is just two FIFO buffers. A 64-word by 48-bit buffer out contains the processor address the link address and the node state fields. A 64-word by 32-bit buffer in contains just the link address and state fields. A PAL controller performs round-robin bus arbitration for the output message bus.

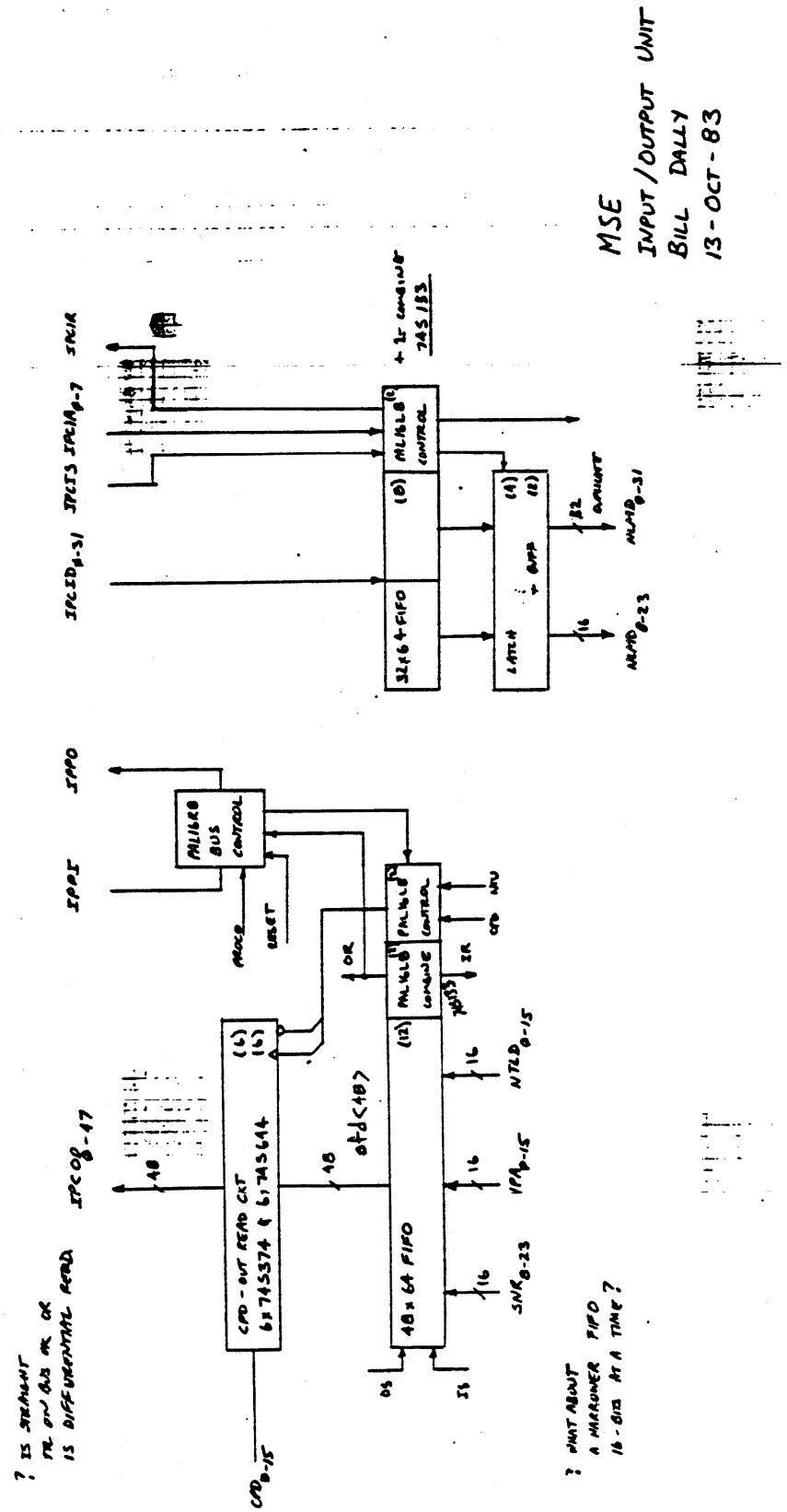


Figure 84. Schematic: Input/Output Unit

6.9 Host Bus Interface

The host bus interface is shown in the following two figures.

Figure 34 shows the MSE side of the host bus. This schematic includes the control processor addressing logic. The AMD29116 microcontroller writes a 24-bit MSE SP address into 74S373 latches shown in the center of the schematic. The low order 16-bits of the address are distributed to the function units for local decoding. The high order 8 bits are decoded by the three DPALs to generate addressing strobes for the function units.

The schematic also shows the statistics counter. This 32-bit counter can be configured so it increments on any one of 16 separate events to log statistics on MSE operation. The number of relaxations performed in any one step (L,P,R or U), the number of scheduling unit pushes or pops in any one step, the number of messages transmitted or received, etc... can be logged. The statistics counter should prove very useful for measuring the performance of various MSE configurations during our experiments with the prototype.

The remainder of the schematic is the actual interface to the host. The host communicates through status and control registers or by taking over control of the SP. A processor address register (PAR) is used to hold the address of the selected processor. A processor must be selected by having its address (8 bits) written into the PAR before it can be accessed by the host. To take over control of the machine, the host sets a bit of the control register which stalls the entire MSE and switches a multiplexer on critical control lines. The CP address registers are floated and the host address buffers are enabled. The host then has direct access to all the MSE memories. Two PALs are used to control host transfers so that writes occur only on ϕ_2.

Figure 35 shows the Multibus interface to the host bus. Because the interface is very simple, only a few bus buffers, it can be easily adapted to other busses.

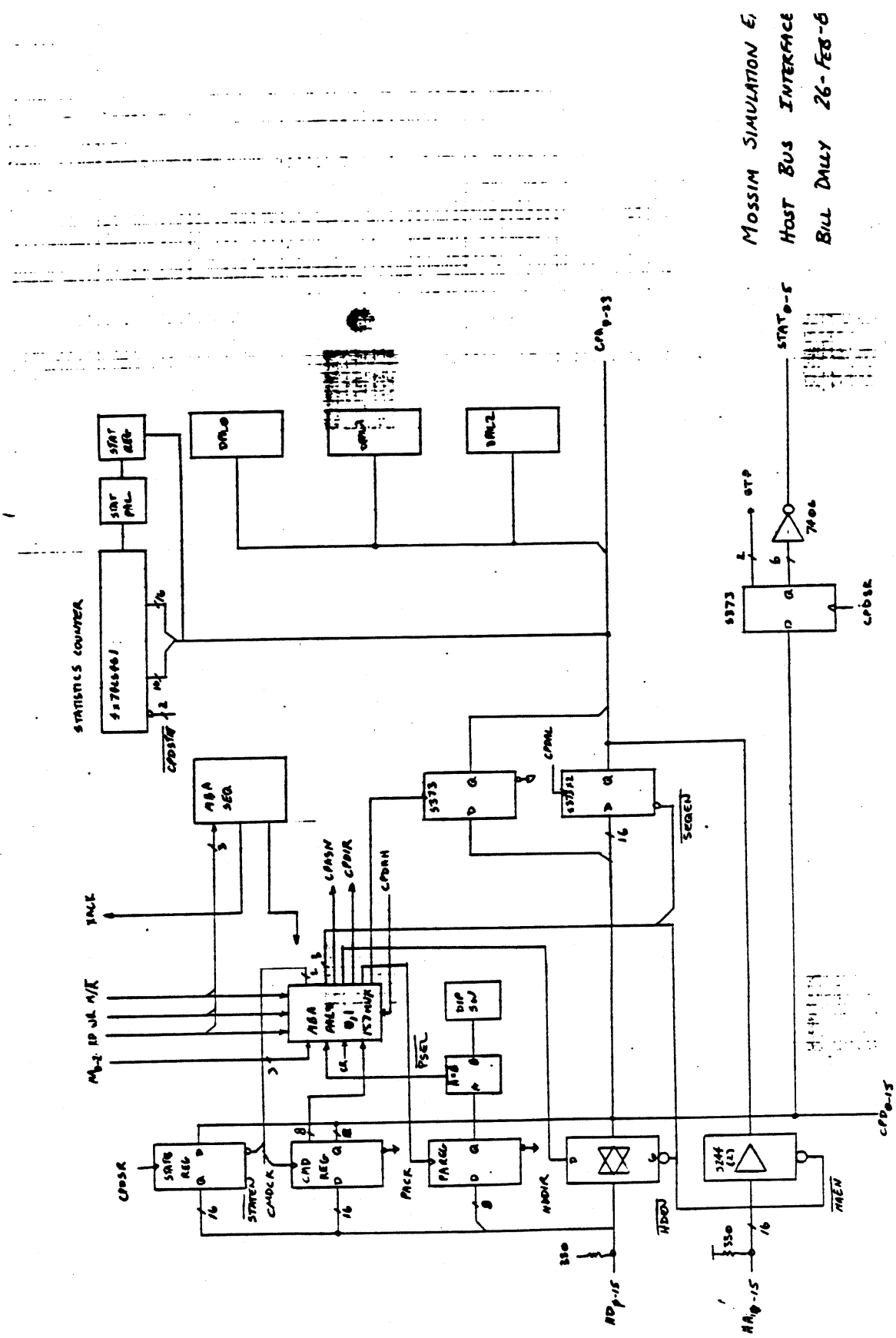


Figure 35. Schematic: Host Bus Interface (MSE side)

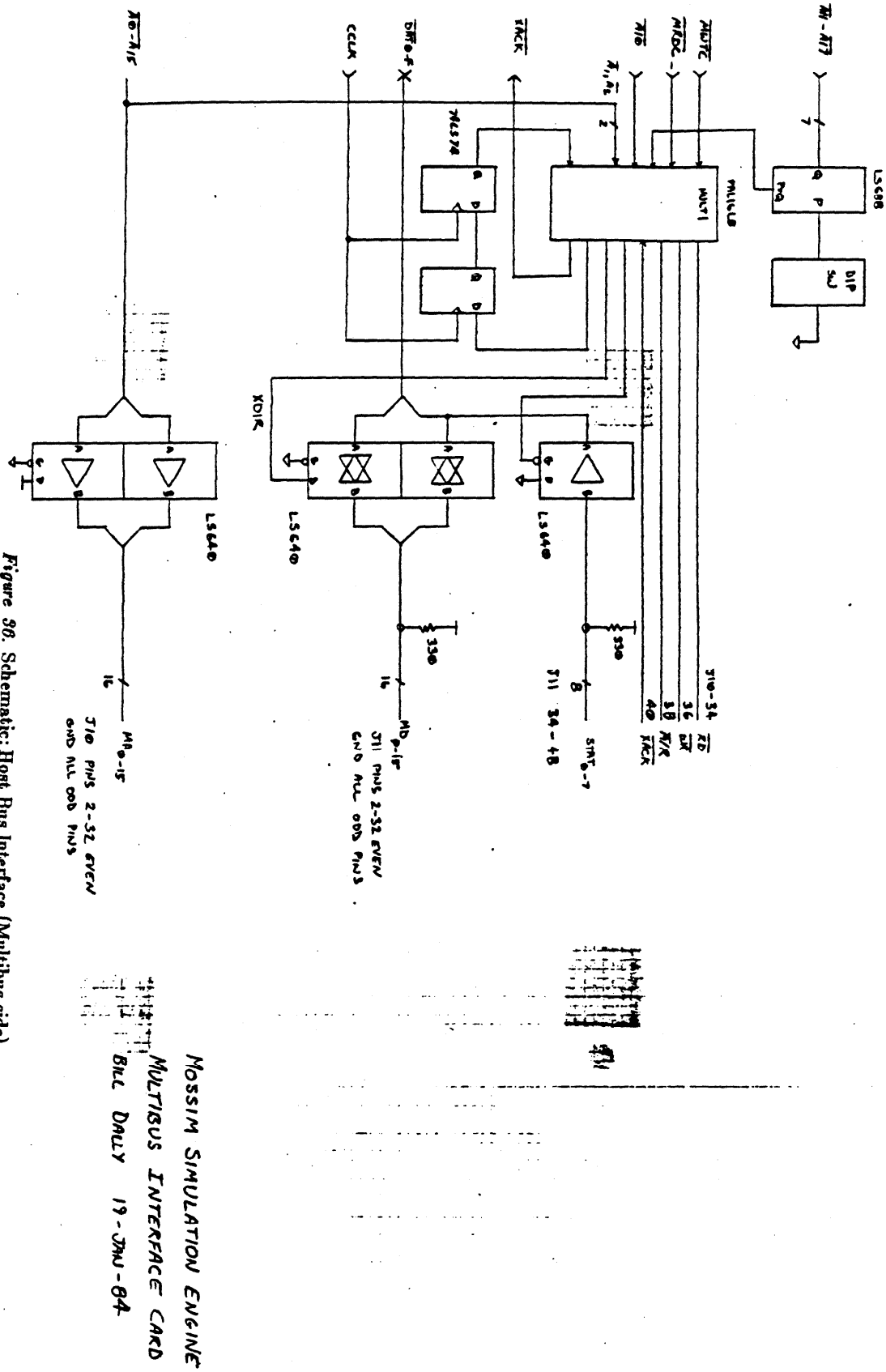


Figure 98. Schematic: Host Bus Interface (Multibus side)

6.10 Conclusion

The design of the MSE required about 700 man hours. During this period of time the ideas presented in this paper when from a jello-like state where things were easily changed to their present concrete state where they are fixed. Now that the design is complete I can look back and see things which I would change if I were to do the design over again. There are also things which I think were done correctly and may be applicable to other special purpose processors.

Things Which I Would Change:

The one thing which will be foremost in my mind the next time I design a digital system will be a firm conviction to keep the design as simple as possible. In a research environment, a special purpose processor is constructed to test an idea. Optimizations not directly related to the idea require a great deal more design time and do not add much to the information gained from the experiment. Also, these optimizations tend to make the machine inflexible. Flexibility is especially desirable in a research machine. In industry, special purpose processors typically have very limited markets, thus the design cost must be amortized over very few units. To make special purpose processors economical, the designer must employ a methodology which minimizes design time possibly at the expense of per-unit cost and performance. The first two items described below are examples of excessive complexity.

Control Logic: A large fraction of the design time went into debugging the local state machines for each function unit, their interactions and the stall logic. While this interlocked pipeline with stalls possibly doubles the performance of the machine, it is not required and the MSE would be just as useful a research instrument without it. In addition to the difficulty of designing this logic, it is also very inflexible compared to a completely microcoded approach to control.

There are many ways this control mechanism can be simplified without sacrificing the performance realized by controlling each function unit with an autonomous state machines. Perhaps the most general design would employ a separate microprogrammed control unit for each function unit.

Timing: Along with a simpler control structure, a revised version of the MSE should do away with the two phase timing described above. A single operation should be performed in each clock cycle and the clock rate should be increased to 10MHz with pipelining added as necessary to achieve this rate.

Local RAM: One of the biggest mistakes in the prototype MSE was leaving off the local RAM. While no chip has been designed at Caltech to date which will not fit into one SP, for large integrated circuits swapping becomes very important. A local RAM would have reduced the overhead of swapping by a factor of 5 since the microcontroller could have transferred one word per clock cycle into the RAM while

one microsecond is required for a Multibus transfer. By incorporating special swapping data paths into the design even higher swapping rates could be achieved.

Custom Chips: The decision to implement the prototype MSE in standard parts was correct. However, now that the design is fairly stable a second implementation could significantly reduce parts count by making use of custom chips to implement some of the function units. In addition units such as the scheduling unit and network traversal unit in a more generalized form may have application to other special purpose processors. One way in which special purpose processors can be economically designed is to build up a library of standard modules which manipulate common data structures and to combine these modules to build a processor.

Things Which were Done Correctly:

In general the design of the MSE is good. It meets its design specifications and follows a regular structure. However, there are a few portions of the design which stand out as being particularly good implementations.

Relaxation Unit: The relaxation unit is an excellent example of what can be done with programmable logic. Implementing a similar unit with standard TTL (no PALs or ROMs) would require ten times as many chips. This design style, however, does not transfer well into VLSI. A ROM is a very poor way of implementing a min-max function on an MOS IC. However, the use of PALs or PLAs and multiplexers does map well to a custom implementation.

Scheduling Unit: The scheduling unit performs its task very well. It can accept up to three commands at once and process them in order of priority. Also, the linked list data structure can easily be adapted to handle other scheduling algorithms. This unit performs a very common function and has applications to other special purpose hardware.

Stall Mechanism: An issue which dominated the performance of the MSE was control latency. A stall mechanism which conditionally halts portions of the MSE pipeline while allowing other portions to continue results in essentially zero latency. A good example of the use of the stall mechanism to achieve low control latency is the interaction of the NOU and NTU. When the NOU needs a link it stalls until the NTU signals that a link is available. The NTU transmits the link immediately since the NOU is already waiting for it. The ready signal itself terminates the stall. No additional delay is introduced into the system by the conditional wait for a link. Other control schemes considered required a one cycle delay to determine the ready condition before proceeding with the stalled operation.

Design Tools:

My experience has been that it takes about twenty times as long to implement a function in hardware as it does in software. The fundamental reason for this time difference is the lack of good hardware design tools. At the start of the project there were really no tools available to support the design of large scale digital systems. The DSIM functional simulator and physical design system was written specifically to support this project. However, DSIM is just an analysis and physical design tool. There are several other areas where good design tools would greatly reduce the amount of design time required.

Control Compiler: In the design of the MSE, more than 75%of the design time was spent on control logic. This time was about equally divided between coding the state machines and the decoder PALs. The state transition diagrams for the state machines and the function of the decoder PALs is not very complex, however a great deal of effort is required to reduce these simple function blocks to logic equations. What is needed is a compiler for digital design which takes a description of the data path of a function unit and a state transition diagram for the controller(s) and generates the control logic. A data path compiler is not really required as data paths are fairly easy to design.

Schematic Entry: The MSE design was coded by hand in the DSIM connectivity language. Schematics were drawn only to visualize the design not as a form of specification. A good schematics entry system would have made entering the connectivity easier. Also, with schematic entry or capture there is only one representation of the design so revisions to the schematics do not have to be copied into the connectivity file and vice-versa.

Chapter 7

Performance

Preliminary performance statistics on the Mossim Simulation Engine have been measured using the MSE Functional simulator to simulate benchmark circuits. These simulations indicate:

- The Mossim Simulation Engine performs switch level simulation at a rate of 250K gate evaluations per second or 200-500 times faster than a VAX-11/780 running MOSSIM-II.
- During simulation the MSE spends about 10%of its time in the logic update step and the remaining 90%of its time equally divided between the perturbation and relaxation steps.
- Only about 5%to 10%of the transistors in a circuit are active at a given time during switch level simulation.

7.1 Functional Simulator

The MSE functional simulator is a 5000 line Mainsail program [34] which simulates the MSE on a clock cycle by clock cycle basis. While the functional simulator does not accurately model the interlocked pipeline, it comes to within $\pm$10%on its timing estimates. The functional simulator was originally written as a tool to evaluate architectural alternatives during the early design of the MSE in August and September 1983. Since that time is has also been used to measure some preliminary statistics on the MSE's performance.

Motivation:

The rationale for building a special purpose processor is that it offers a significant improvement in performance over general purpose machines. Thus, it is important in devesloping an architecture to

have a quantitative measure of performance with which the merit of competing architectures can be judged. Too often the architecture of a digital system is arrived at in an ad-hoc manner depending on the designer's intuition to decide between alternative approaches. For the most part the architecture of the MSE was arrived at by simulating alternative approaches using a functional simulator.

Simulation Module:

The functional simulator is composed of three mainsail modules: simulator, user interface and loader. The simulator module performs the actual simulation using the MOSSIM algorithm adapted for hardware described in chapter 3. Mainsail data structures are used to represent the memories and registers of the MSE and procedures represent each operation which can be performed on a data element. Each operation charges time to the simulation depending on the current state of the machine. Higher level procedures implement the algorithm by calling the low level procedures to operate on the MSE data structure.

Loader Module:

The loader module is a prototype for a loader which will eventually support the actual MSE. It takes an input file in NTK format [11] and produces an MSE memory image. This memory image is then loaded by the simulation module.

The loader module operates in four steps.

1. The NTK file is read in and a network data structure is built. This data structure is identical to the MSE data structure: nodes, gates, and links; however it covers the whole network in one structure and must be partitioned into VP sized subnetworks.

2. The first step of network partitioning is performed by finding transistor connectivity groups (described in chapter 4), and constructing a directed graph of these connectivity groups with the edges weighted by the number of gate connections between two groups.

3. Using the data structure produced in step 2 the network is divided into VP sized clusters using a heuristic graph partitioning algorithm.

4. Code is generated for the partitioned data structure and written into an MSE memory image file.

One problem with the existing loader is its speed. Some ideas for fast loaders are discussed in chapter 8.

User Interface Module:

The user interface module provides a set of MOSSIM-like commands for controlling simulation. Nodes can be set and watched, or grouped into vectors etc.... Only a subset of the commands supported by MOSSIM-II [11] are implemented. The user interface also includes a monitor which allows the user to examine or change internal registers and memory in the simulated MSE.

7.2 Throughput

To determine the throughput of the MSE in terms of gate evaluations per second, four benchmark counter circuits were simulated on the MSE functional simulator. The number of clock cycles and number of logic events were recorded for all four circuits approximately 20 clock cycles were required for one logic event giving a throughput of 250K logic events per second.

The detailed statistics are presented below. The abbreviations each contain two letters. The first letter specifies the simulation step and the second letter gives the type of statistic.

N measures a cumulative internal variable and is an indication of the amount of time spent in a particular simulation step;

E represents the number of events (source nodes popped);

C represents the number of relaxation or update cycles (destination nodes examined).

The variables TLE and TCY are the total number of logic events and the total number of clock cycles respectively. The ratio CY/E gives the throughput. The bottom line of statistics for each test gives the total number of clock cycles spent in each simulation step. These statistics are discussed in the next section.

```
Test 1: 16-bit standard counter clocked for 150 counts

Proc: 0 LN = 4818 LE = 32 LC = 65
PN = 20075 PE = 51 PC = 228
RN = 31607 RE = 52 RC = 198
UN = 38124 UE = 52 UC = 214

TLE = 18704     TCY = 354327     CY/E = 18
Totals: 39670   101509  102942  110206
```

```
Test 2: 32-bit counter for 150 counts

Proc: 0 LN = 5428 LE = 64 LC = 129
PN = 35256 PE = 115 PC = 484
RN = 55927 RE = 116 RC = 422
UN = 67468 UE = 116 UC = 454

TLE = 31088      TCY = 616225     CY/E = 19
Totals: 64808    178786  179790  192846

Test 8, 32-bit Manchester carry chain counter

Proc: 0 LN = 50070 LE = 64 LC = 129
PN = 182285 PE = 170 PC = 767
RN = 213806 RE = 197 RC = 783
UN = 253472 UE = 197 UC = 815

TLE = 134608     TCY = 2638619    CY/E = 19
Totals: 281070   744059  784769  828721

Test 4 - 4-bit counter for 150 counts

Proc: 0 LN = 3873 LE = 8 LC = 17
PN = 8198 PE = 9 PC = 50
RN = 12503 RE = 10 RC = 44
UN = 15177 UE = 10 UC = 48

TLE = 8552       TCY = 147631     CY/E = 17
Totals: 18916    41084   42393   45238
```

7.3 Load Distribution

The load distribution over simulation steps in the MSE, derived from the number of clock cycles spent

in each step, is shown in figure 37 below. Only 10%of the time is spent in the logic update step while over 60%of the total time is spent in the relaxation steps. The design of the MSE has been influenced by these statistics with an emphasis being placed on improving the performance of the relaxation process. In the future, we believe that the most performance improvement will result from optimizations in the perturbation step which will reduce the number of nodes which must be re-evaluated during the relaxation steps.

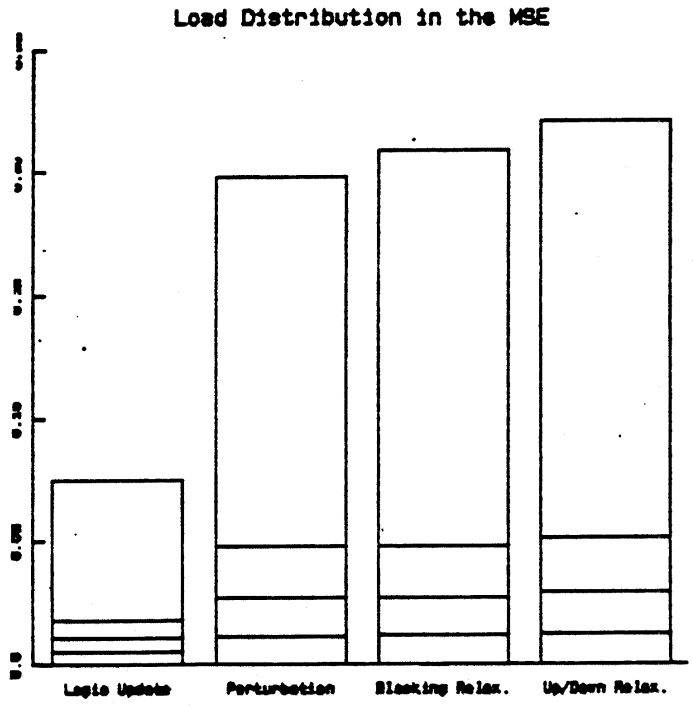


Figure 37. MSE Load Distribution

7.4 Activity

In section 4.3, measurements of the locality of activity in three benchmark circuits were presented. These results, obtained through functional simulation, show that the number of logic events which occurs during a cycle is typically between 5%and 10%of the number of transistors in the circuit.

These measurements may actually be higher than the percentage of active circuitry. Often a single

transistor in an active portion of the circuit changes state several times in one cycle.

An important question which simulations on the prototype MSE should help to answer is how this activity is distributed when the network is partitioned into subnetworks. If 5%of the subnetworks have 100%activity the working set of virtual processors will be very small and a small physical MSE would simulate a large circuit with little degradation. However if the activity does not partition but is uniformly distributed over subnetworks swapping performance will be very bad.

7.5 Conclusion

Using the MSE functional simulator estimated throughput, load distribution and activity of the MSE SP have been measured. Several important performance parameters, however, still need to be determined. Most of these parameters have to do with the performance of the MSE in a multiprocessor environment. While the MSE functional simulator can simulate multiple MSE SPs, only small circuits could be simulated because of the limited memory on the Caltech Computer Science DEC 20/60 on which these simulations were run. To gain insight into multiprocessor performance, large circuits must be simulated.

When the prototype MSE is constructed several experiments to determine multiprocessor performance are planned:

- Message traffic for circuits of various sizes and varying VP sizes will be measured.
- The locality of activity within a subnetwork will be measured. It is believed that activity is concentrated within small areas of the network so the level of activity in an active subnetwork should be much higher than the overall activity of the network.
- Swapping performance will be measured for various circuits and various working set sizes.

Chapter 8

Circuit Compilation and I/O Vectors

The use of special purpose simulation hardware increases the performance of simulation to the point where supporting functions such as circuit compilation and the transfer of I/O vectors to and from the simulation engine can become bottlenecks. The simulation becomes in effect I/O bound. In this chapter algorithms for fast circuit compilation and a means of using functional simulation to generate I/O vectors are described.

Simulators are used in two modes: debugging and regression. In regression mode a large battery of tests is run against a circuit with little interaction from the user. For example during the design of a processor with 150K transistors 300 tests containing over 1000 vectors each were run on each version of the design. These regressions were performed to assure that no new bugs were introduced into the design. Since the circuit is loaded only once during a long regression, circuit compilation performance is not important in this mode of simulation. I/O vector rate however is very important for regression performance.

In debugging mode, a designer uses the simulator to evaluate small design changes. A typical design iteration is illustrated in figure 38 below. The designer changes the network description by editing a few cells or changing a few connections between cells. Generally the changes are very minor and the network is almost identical to the previous version. The new circuit description is then partitioned into VP sized subnetworks and compiled into MSE memory images. The MSE loads these images and performs the simulation. In debugging mode the simulations are generally short and highly interactive with the user stopping the simulator frequently to examine the state of nodes or to change the input vectors. Since very little simulation is performed the circuit compilation can become a bottleneck. Since only a small part of the circuit changes each iteration, an incremental method of circuit compilation would greatly reduce the amount of time spent in this step.

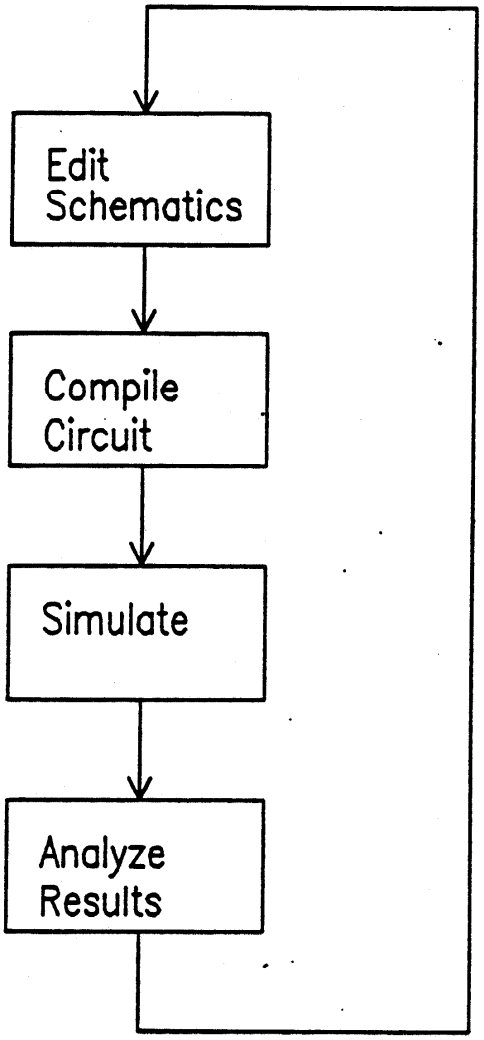


Figure 38. Design Iteration Cycle

8.1 Circuit Compilation

Recompiling an entire MOS circuit each time a small change is made is a waste of time. In this section an incremental algorithms for circuit compilation is presented.

The current version of the MSE functional simulator (described in chapter 7) requires a considerable amount of time to parse a *flat* network description and to partition this network among MSE processors. In an interactive simulation environment, this long compile time is objectionable. One method of overcoming this problem is to use a hierarchical partitioning algorithm such as described in [40]. With this approach, the network is expanded from the hierarchical description and partitioned simultaneously. During the expansion, the natural boundaries between the hierarchical cells are used to guide the partitioning. Hierarchical compilation reduces the amount of time spent in partitioning the network.

Even with a hierarchical algorithms, the entire network must still be recompiled each time a change is made. An incremental algorithm is required to reduce compile time to a level proportional to the number of changes made.

To implement incremental compilation each leaf cell is compiled into a relocatable module as shown in figure 39 below. Each module is organized so that the interface portions of the circuit description do not change location when the circuit is modified. All nodes which are ports of the cell are placed at the beginning of the node list. Similarly all links and gates which can be referenced by external cells are placed at the beginning of the link and gate lists. The positions of these interface elements are fixed, so addresses in other cells which point at this cell need not be updated when the cell is changed. Only the completely internal circuit elements are changed. To allow circuitry to be added to the cell without increasing its size, extra space is left at the end of the cell.

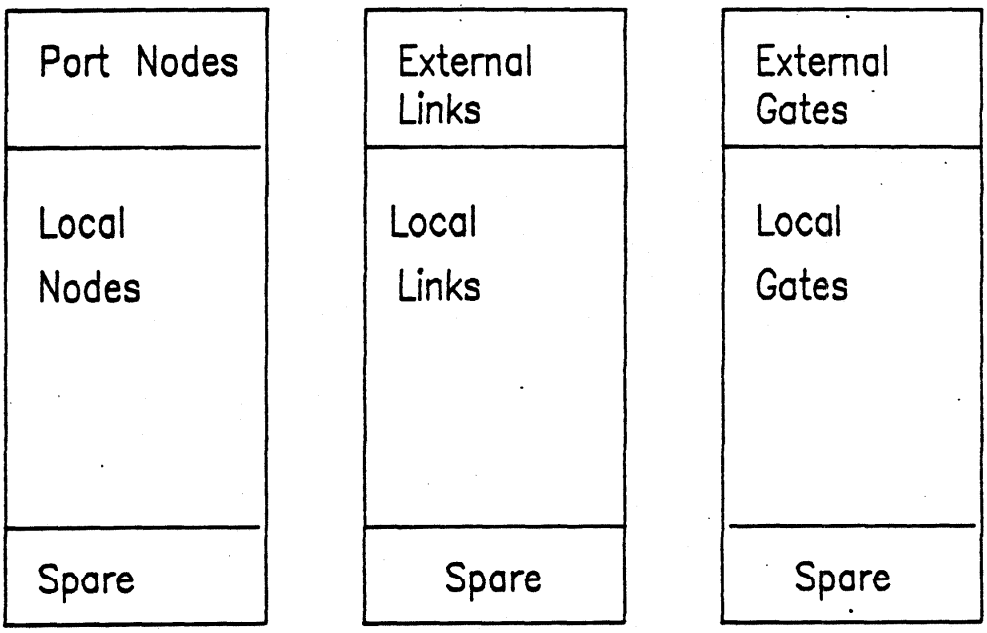


Figure 39. Relocatable Circuit Module

Using a hierarchical partitioning algorithm these relocatable leaf cell modules are combined to form higher level cells by concatenating their node link and gate lists and then linking their external pointers. Since the locations of leaf cells and the interface addresses of each leaf cell are fixed, a leaf cell can be changed without altering the rest of the circuit as shown in figure 40 below. The designer enters the changes to the leaf cell into the design data base. The cell is compiled into a new relocatable module with the same interface and size as the old module. The new module body is then copied into the MSE memory image for each instantiation of the leaf cell.

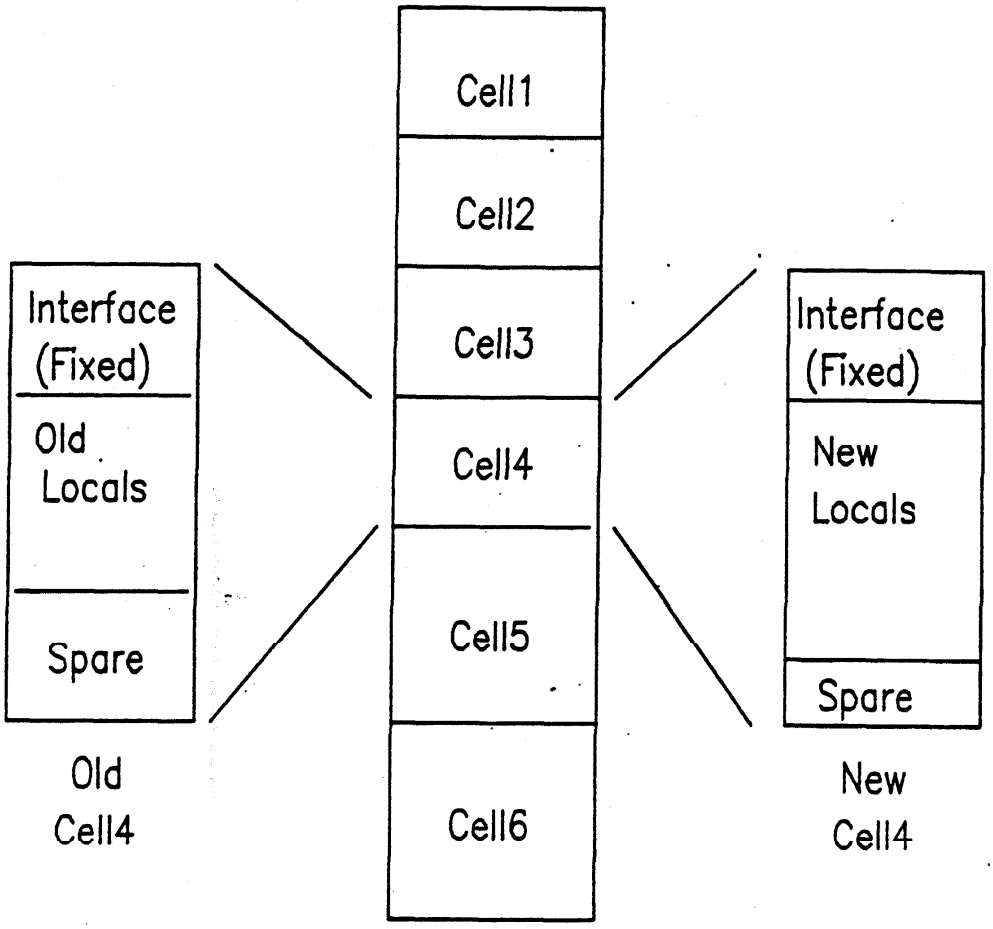


Figure 40. Incremental Update of a Leaf Cell

When connections within a higher level cell are changed, the circuit compiler attempts to recompile the high level cell so that it also has fixed interface addresses and fixed size. However, for some high level cells it may be impossible to maintain a constant interface and the entire circuit may have to be recompiled and linked.

The incremental circuit compilation algorithm described above borrows from software engineering the concepts of relocatable code, linking and separate interface and implementation sections for modules. This modular circuit model is then imposed on the structure of the MSE which is inherently flat. In a future version of the MSE, additional hardware could be added to support a modular circuit model.

Such hardware would allow circuit models to reference ports of cells by cell name and port name rather than by physical address. A virtual to physical translation of the port address to physical node, link or gate address would be performed in effect implementing dynamic linking of cells. Module linkage and management implemented in hardware would make incremental circuit compilation a great deal simpler.

8.2 Functional Generation of I/O Vectors

For some types of simulation the MSE can become I/O bound as the host cannot transfer I/O vectors to/from the subnetwork processors fast enough to keep them busy. Functional simulation in an AP can be used to generate input vectors and verify output vectors without involving the host. The AP holds a functional model of the circuit's environment and generates logic events directly to update input pins. For output, the AP can check pertinent output pins against the expected response.

Functional simulation has the additional advantage that it generates input vectors and checks output vectors procedurally. This procedural I/O can be used to generate timing independent I/O vectors. Consider for example a microprocessor which expects data to be placed on its data bus when an output line goes low. Suppose the microprocessor is modified so an addition clock cycle is inserted before each memory read. Using the traditional method of inputting a specific test vector each cycle, the input vectors would no longer be in sync with the processor. Functional simulation on the other hand could be programmed to wait for the data strobe before intputting the data vector and would automatically synchronize itself with the processor.

Simulation environments can also be created using functional simulation. For example, in the case of the microprocessor described above an external memory could be implemented using functional simulation. The processor could then be simulated by loading programs into this memory and executing them.

Chapter 9

Status Report

Status: The MSE project was begun in July, 1983. The following has been accomplished to date:

1. A study of static MOS network locality has been performed. The results of this study have been used to estimate message switch traffic in the MSE.

2. The current state-of-the-art in simulation engines has been surveyed. The results of this survey are presented in chapter 2.

3. The MSE architecture presented in chapter 5 of this paper was developed. The architecture was motivated by: the MOSSIM algorithm, the architecture of existing simulation engines, and the locality properties of MOS networks.

4. A functional simulator (accurate to the register transfer level) of a single processor MSE has been implemented to verify the architecture. This simulator is described in chapter 7.

5. Simulations of the MSE architecture have been performed to measure throughput, load distribution and activity.

6. The DSIM functional simulation system was developed to allow the design to be simulated at the chip level.

7. An implementation of the MSE architecture in standard TTL and MOS parts was developed and coded in the DSIM connectivity language.

8. A DSIM physical design package has been implemented to generate a file suitable for driving an automatic wire-wrap machine to support construction of the MSE.

Plan: Now that the design of the MSE is complete, experiments to test the machine and to determine its performance in a multi-processor environment are being planned. First however, the machine must be constructed, debugged and support software must be written. The project plan is as follows:

1. The MSE will be constructed by June 1984

2. Debugging is expected to be completed by August 1984

3. Support software for the MSE will be developed while the machine is being debugged. A monitor for the machine allowing memories to be displayed and edited, the machine to be single stepped etc... will be written to support debugging. An incremental circuit compiler as described in chapter 8 will be written to load circuits in either NTK or SIM format into the machine. Also, a user interface will be written to support a subset of the MOSSIM-II [11] commands for interacting with the simulator. A graphics oriented user interface is also under consideration. No date has been set for completing the software.

4. Once the software is operational, the experiments described in chapter 7 for measuring parameters relating to multiprocessor MSEs will be performed.

5. If the project continues, the MSE message switch and auxiliary processor will be designed and constructed.

Chapter 10

Conclusion

The MOSSIM Simulation Engine, MSE, is a special purpose processor designed to efficiently implement the MOSSIM algorithm [10]. This paper has described the algorithms and architecture of the MSE. Five objectives presented in the introduction motivated the MSE architecture. In section 10.1, we will see how each of these objectives has been met. The MSE is intended to be a dedicated special purpose processor, however, its hardware is suited for a family of tasks involving network manipulation. In fact, the basic MSE architecture is suited for a much wider range of applications involving functional programming. These applications are discussed in section 10.2. The report concludes with a discussion of other special purpose processors.

10.1 Objectives

The MSE achieves each of the five objectives presented in the introduction.

1. *Accurate Simulation:* The MOSSIM algorithm provides accurate simulation of MOS circuits at the switch level.

2. *Performance:* A single processor MSE is estimated to improve the speed of simulation by a factor of between 200 and 500 compared with MOSSIM running on a DEC VAX 11/780. Multiprocessor MSE configurations should improve this throughput nearly linearly with the number of processors.

3. *Scaling:* The functional simulation supported by the MSE allows large circuits to be simulated by suppressing unnecessary detail. With the virtual processor and hierarchical addressing features, the MSE can handle extremely large circuits at the transistor level.

4. *Multi-Level Simulation:* is supported by the MSE's auxiliary processors.

5. *Extensibility:* The MSE can be extended in capacity by inserting additional SPs. Features can be added through microcode.

10.2 Applications

While the MSE is optimized to execute the MOSSIM algorithm, it provides a framework for general network manipulation. The only part of the MSE specialized to MOSSIM is the relaxation unit (RU) in the node operation unit (NOU). The NTU provides a general set of capabilities for traversing a network database. These primitives can be used for applications including circuit simulation, timing simulation, electrical rules checking, etc.... Similarly, the SU provides a general purpose scheduling capability. While four processing steps are scheduled for the MOSSIM algorithm, the SU can provide for any number of steps and for diverse scheduling algorithms.

The basic architecture of the MSE provides a framework for general functional programming. The networks which can be processed by the MSE are not limited to MOS transistor networks. The tasks of scheduling and graph traversal are universal to a number of applications. If the graph to be traversed is a dataflow graph, the MSE architecture can be applied to functional programming. Consider a variant of a dataflow machine where an operator executes any time one of its operands changes. This type of machine executes a functional program. The relation between the variables in the program is fixed. Any time one variable changes, all variables dependent on this variable must be recalculated. Such a machine is easily mapped onto the MSE architecture. When a variable changes, the NTU generates the list of all dependent variables which must be recalculated. If any of these variables changes as a result of the calculation, the SU performs scheduling for variable update. The only portion of the machine which would change significantly is the NOU. A more general set of operations would replace the relaxation operations performed in the MSE. A data-driven machine of this type would have applications in computer graphics, finite element analysis and real-time control systems.

10.3 Special Purpose Processors

Simulation is just one example of a computer application which has outgrown the capabilities of general purpose computers. Other applications in the field of design automation include sticks compaction, placement, routing, circuit extraction and design rule checking. Dozens of applications in other fields are also to the point where they require special purpose hardware.

The methodology followed in developing hardware for switch-level simulation can be applied to other applications as well.

1. The problem is studied to measure statistics which influence the cost and/or performance of the algorithm. For the MSE, the locality of MOS networks was studied.

2. The algorithms to be implemented are modified as necessary to fit into a hardware implementation. Often entirely new algorithms may be required. For hardware implementation algorithms must involve repeatedly performing relatively simple operations.

3. The algorithm is examined for potential concurrency and for applications of specialization, and the algorithm is partitioned into hardware modules. Some candidate architectures are developed. Step 2 may have to be repeated if the algorithm does not exhibit enough potential concurrency for performance improvement.

4. Performance metrics for the problem are developed and candidate architectures developed in step 3 are evaluated in terms of these metrics. As analysis proceeds the architecture is refined.

5. A technology is selected and the architecture developed in step 4 is implemented.

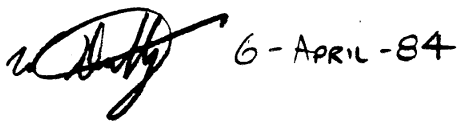
The rapid advance of integrated circuit fabrication technology has made it possible to construct special purpose hardware at a reasonable cost. However, to economically develop large digital systems of this nature, better design methodologies and supporting software are required. Currently, the digital systems designer spends the majority of his time working out details of the design rather than developing concepts. This low level hacking is analogous to writing machine code programs. Too much of the current design automation effort is concentrated on low level tools associated with design verification and physical design. High level tools are urgently needed to make the development of custom hardware for many applications economically feasible.

Along with high level tools, higher level functional building blocks are required. The functional modules in special purpose processors typically involve manipulating data structures such as stacks, queues, and graphs. Functional building blocks for manipulating these data structures should be developed and standardized so they can be rapidly integrated into special purpose systems.

10.4 Acknowledgements

I would like to thank Randy Bryant and Chuck Seitz for their support and guidance during the MSE

project. Discussions with several members of the Caltech community: Bill Athas, Mike Schuster, and Dan Whelan, helped in the development of the MSE. Special thanks goes to Bill Athas for reviewing a draft copy of this manuscript. I would also like to thank my wife, Sharon, for her support of my graduate studies and encouragement in pursuing this research.



Chapter 11

References

[1] Moore, Gordon, "VLSI: Some Fundamental Challenges," *IEEE Spectrum*, April 1979, pp. 30-37.

[2] Johannsen, David, *Silicon Compilation*, Ph.D. Thesis, Caltech Technical Report 4530, 1981.

[3] Pfister, G. F.,"The Yorktown Simulation Engine: Introduction," *19th Design Automation Conference*, ACM, 1982, pp. 51-54.

[4] Denneau, M. M.,"The Yorktown Simulation Engine," *19th Design Automation Conference*, ACM, 1982, pp. 51-54.

[5] Kronstadt E. and G. Pfister,"Software Support for the Yorktown Simulation Engine," *19th Design Automation Conference*, ACM, 1982, pp. 51-54.

[6] "ZyCad LE-001 and LE-002 Product Description," ZYCAD,1982.

[7] Bartow, R. L. *et al.*,"Architecture of a Hardware Simulator," *IEEE Conference of Circuits and Computers*,1980, pp.891-893.

[8] Bryant, R., *A Switch-Level Simulation Model For Integrated Logic Circuits*, Ph.D. dissertation, MIT, 1981.

[9] Bryant, R., "MOSSIM: A Switch-Level Simulator For MOS LSI," *18th Design Automation Conference*, ACM, 1981, pp. 786-790.

[10] Bryant, R., *A Switch-Level Model and Simulator for MOS Digital Systems*, Caltech Technical Report 5065:TR:83, January 1983.

[11] Bryant, R., M. Schuster and D. Whiting, *MOSSIM II: A Switch-Level Simulator for MOS LSI,*

User's Manual, Caltech Technical Report 5033:TR:82, January 1983.

[20] Szygenda, "TEGAS-2 – Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic," *9th ACM-IEEE Design Automation Workshop,* 1972, pp. 116-127.

[13] Chawla, B., H. K. Gummel, and P. Kozah, "MOTIS - an MOS Timing Simulator," *IEEE Transactions on Circuits and Systems,* vol. CAS-22, December 1975, pp. 901-910.

[14] Vladimirescu, A. *et al., SPICE Version 2G.5 User's Manual,* University of California, Berkeley, Technical Memo.,August 1981.

[15] Dally, W. and R. Bryant, *The Mossim Simulation Engine, Preliminary Architecture,* Caltech Display File 5100:DF:83, September 1983.

[16] Dally, W. and R. Bryant, *The Mossim Simulation Engine,* unpublished paper, November 1983.

[17] Howard, J. R. Malm and L. Warren, "Parallel Processing Interactively Simulates Complex VLSI Logic," *Electronics,* December 15, 1983, pp. 147-150.

[18] Breuer, M. A. and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems.,* Computer Science Press, 1976.

[19] Barzilai, Z. *et al.,* "Simulating Pass Transistor Circuits Using Logic Simulation Machines," *20th Design Automation Conference,* ACM, 1983, pp. 157-163.

[20] Szygenda, "TEGAS-2 – Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic," *9th ACM-IEEE Design Automation Workshop,* 1972, pp. 116-127.

[21] Abraomovici, M. et al.,"A Logic Simulation Machine," *19th Design Automation Conference,* ACM, 1982, pp. 65-73.

[22] Abramovici, M. et al.,"A Logic Simulation Machine," *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems,* Vol CAD-2, No.2, April 1983, pp. 82-94.

[23] Lin, T. and Mead C. A., *Signal Delay in General RC Networks with Application to Timing Simulation of Digital Integrated Circuits,* Caltech Technical Report 5089:TR:83, July 1983.

[24] Dally, W., *The DSIM Functional Simulation System: Preliminary User's Manual,* Caltech Display File 5109:DF:83, December 1983.

[25] Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms,*

Addison-Wesley, 1974.

[26] Ousterhout, J.K.,"Crystal: A Timing Analyzer for nMOS VLSI Circuits," *Proceedings of the 3rd Caltech VLSI Conference*, R. Bryant Ed., Computer Science Press, March 1983, pp. 57-70.

[27] Jouppi, N.P., "TV: An nMOS Timing Analyzer," *Proceedings of the 3rd Caltech VLSI Conference*, R. Bryant Ed., Computer Science Press, March 1983, pp. 71-86.

[28] Agrawal, V. D., "Synchronous Path Analysis in MOS Circuit Simulator," *19th Design Automation Conference*, ACM, 1982, pp. 629-635.

[29] Adelfio, S., P. Agrawal and W. Dally, *DATUM - Delay Analysis Tool Using MOTIS*, Bell Labs Technical Memorandum, 1981.

[30] Ruehli, A. E., and G. S. Ditlow, "Circuit Analysis, Logic Simulation, and Design Verification for VLSI," *Proceedings of the IEEE*, Vol. 71, No. 1, January 1983, pp. 34-48.

[31] Mattisson, S.,"*MOS-VLSI Circuit Simulation Formulations for Concurrent Execution*, Caltech Display File 5096:DF:83, August 1983.

[32] Newton, A. R., "Techniques for the Simulation of Large-Scale Integrated Circuits," *IEEE Transactions on Circuits and Systems*, vol. CAS-26, September 1979, pp. 741-749.

[33] Hill, D. D., and W. M. VanCleemput, "SABLE: Multilevel Simulation for Hierarchical Design," *Proc. IEEE Symposium on Circuits and Systems*, April 1980, pp. 431-434.

[34] Wilcox, C. R., M. L. Dageforde, and G. A. Jirak, *Mainsail (TM) Language Manual Version 4.0*, XIDAK, 1979.

[35] Dally, W., *Locality of MOS Circuits*, unpublished locality statististics, July, 1983.

[36] Lutz, C. *et. al.* "Design of the MOSAIC Element," *1984 Conference on Advanced Research in VLSI*, MIT, Artech, January 1984, pp 1-10.

[37] Mandelbrot, B., *Fractals: Form, Chance and Dimension*, pp. 237-239, W. H. Freeman, SanFrancisco, 1979.

[38] Landau, B. and R. Russo, "On a Pin vs Block Relationship for Partitions of Logic Graphs," *IEEE Transactions on Computers*, vol. C-20, pp 1469-1479.

[39] Denning, P., "The Working Set Model for Program Behavior," *Communications of the ACM,*

Vol. 11, No. 5, May 1968, pp.323-333.

[40] Payne, T. S. and W. M. VanCleemput, "Automated Partitioning of Hierarchically Specified Digital Systems," *19th Design Automation Conference*, ACM, 1982, pp. 182-192.